Verifying Security Properties in Electronic Voting Machines

by

Naveen K. Sastry

B.S. (Cornell University) 2000

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Spring 2007

The dissertation of Naveen K. Sastry is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2007

Verifying Security Properties in Electronic Voting Machines

Abstract

Verifying Security Properties in Electronic Voting Machines

by

Naveen K. Sastry

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Wagner, Chair

Voting is the bridge between the governed and government. The last few years have brought a renewed focus onto the technology used in the voting process and a hunt for voting machines that engender confidence. Computerized voting systems bring improved usability and cost benefits but also the baggage of buggy and vulnerable software. When scrutinized, current voting systems are riddled with security holes, and it difficult to prove even simple security properties about them. A voting system that can be proven correct would alleviate many concerns.

This dissertation argues that a property based approach is the best start towards a fully verified voting system. First, we look at specific techniques to reduce privacy vulnerabilities in a range of voting technologies. We implement our techniques in a prototype voting system. The componentised design of the voting system makes it amenable to easily validating security properties. Finally, we describe software analysis techniques that guarantee that ballots will only be stored if they can later be accurately reconstructed for counting. The analysis uses static analysis to enable dynamic checks in a fail-stop model.

These successes provide strong evidence that it is possible to design voting systems with verifiable security properties, and the belief that in the future, voting technologies will be free of security problems.

Professor David Wagner
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful for David Wagner's insightful input in crafting this dissertation. His influence permeates each section and I am fortunate to have such a caring advisor. He patiently taught me the basics and listened to my asinine and ill-informed ideas. He removed obstacles and served as a great model to follow, always humble and kind. I learned not only research from him, but also ethics, honesty, and character.

I have long joked that I would show up in my colleagues' dis-acknowledgments for slowing down their progress. Fortunately, I can safely say that my colleagues were kinder to me and became my friends, and made work fun. They refined my ideas and improved my research quality. Umesh Shankar taught me many of the paper-writing basics in one of my first papers and continued to hone my ideas. Chris Karlof has been a frequent co-author, sounding board, and constant friend. Along with Chris, Adrian Mettler and Yoshi Kohno each were crucial co-authors on papers that formed the basis of this work. Manu Sridharan was not only a gym-buddy, but also a helpful resource for all my PL questions. Finally, I will fondly reminisce about my days in 567, as I discussed Economics, women, and more with Rob Johnson and Karl Chen.

I want to thank my parents, sister, and family for their loving support. Their sacrifices gave me the opportunity, tools, and especially the confidence to tackle graduate school.

And finally, my tremendous wife, Seshu, deserves my eternal gratitude. She endured practice presentations and editing, while soothing my frustrations in completing the dissertation. Her gentle encouragements and patient understanding were crucial to finishing on time.

Chapter 1

Introduction

1.1 The voting problem: motivation

The 2000 Presidential election brought attention to the importance of accurately recording and tabulating ballots, and a hunt for new technologies to fix the unearthed problems. Election officials faced considerable difficulty deciphering voters' selections. *Direct Recording Electronic* (DRE) voting machines were seen as one solution and are now deployed in many counties. These computerized machines offer advantages over traditional lever, paper, or punch card voting systems. They eliminate classes of ballot marking errors using software logic to rule out voting for multiple candidates where only one is allowed, for example. Since voters interact with a computer screen, the DRE machines can adopt the interface that best suits the needs of a voter. For example, they can switch to large, high-contrast fonts for voters with reduced visual acuity. Additionally, tabulating the results is quicker than with other systems since each machine effectively maintains a running sum.

However, the advantages that current DRE systems offer do not come without risk. DREs are built upon general purpose computers, and are designed with standard software development techniques. Standard software development techniques often lead to code that is buggy and suffers from latent vulnerabilities. Voting software is no different: Kohno et al. recently performed a security audit and showed the software on these machines is not well designed and riddled with severe security bugs [42]. This study is not unique in its conclusions, as others have found innumerable security problems in commercial voting code [18, 25, 72, 90, 94].

Currently deployed DREs use a single monolithic application written in an unsafe language, such as C. Unless great care is taken, software written in C can suffer from buffer overruns, improper type coercions, and programmer errors that lead to memory safety violations. In addition, the software is just too complex to be sure all security bugs can be eliminated even with a careful audit. This naturally begs the question: can we do better?

One option is for counties to deploy non-DRE based voting technology, of which there are several options, such as optical scan readers. But given the prevalence and advantages of DREs, it is necessary to address their shortcomings. In this dissertation, we focus on DREs. Thus far, voting and security experts have come to two potentially viable remedies to sidestep the issue of buggy voting software in DREs. Both approaches are designed to detect voting machine errors and still yield the proper election tally.

In the first, DREs are augmented with printers to produce paper records of the voter's choices. Before leaving the voting booth, the voter checks the printed record accurately represents their choices [50]. This voter verified paper audit trail (VVPAT) can serve as an official recourse in case the electronic record is suspect.

Alternatively, C. Andrew Neff and David Chaum have each come up with innovative solutions that rely on cryptography [19, 60, 61]. After voting on a traditional DRE, their systems engage in a cryptographic protocol with the voter. During the protocol, the DRE prints a specially formated receipt. The receipt does not reveal any information about the voter's choices, but it does allow the voter to take the receipt home and verify their vote hasn't been changed after they voted and that their vote will be counted. This property, called universal verifiability, is unique to cryptographic voting protocols.

Both solutions offer advantages over existing DREs, however in this dissertation we show that those two solutions are not sufficient since there are classes of privacy violations left unaddressed. We also propose new techniques that begin to address their shortcomings in DRE based voting machines.

1.1.1 Approach

The solutions we pursue are aimed at one central goal: simplifying an auditor's task in verifying the correctness of security properties in voting machines. This is distinct from another, perhaps more obvious goal: eliminating security bugs from voting machines. While the latter goal is more appropriate for many software applications, it is not sufficient for the voting context. As Dan Wallach has said, "The purpose of an election is not to name the winner, it's to convince the loser they lost" [85]. Consequently, it is not enough to eliminate all security bugs: we must develop ways for interested third parties to verify for themselves that the voting machine is free of security bugs.

Making it easier to verify the absence of security bugs is particularly relevant given that voting machines currently receive little oversight. Counties rely on a handful of *Independent Testing Authorities* (ITAs) to ensure that a vendor's voting machine complies with voting standards and meets nominal security requirements. In one study, we found 16 critical security vulnerabilities in Diebold voting code [90], while CIBER, an ITA given the same mandate to evaluate the same code, produced a vastly different report and only found three security vulnerabilities [21]. This contrast highlights the main motivation for this work: to help auditors and citizens verify that their voting system is secure.

In verifying a voting system, an auditor or concerned citizen must analyze a voting system against a set of measurable criteria. For example, one such criterion may be that a voting system always gives the voter a chance to review their ballot and correct any mistakes they discover before casting. We call these measurable criteria *properties*. The Voluntary Voting System Guidelines produced by the United States Election Assistance Commission is one such list of these properties. These properties are created to reflect societal goals, norms, and laws with respect to voting. Since goals can often be vague, it is important to have a precise definition of what is being verified. Properties are meant to embody this greater specificity and measurability. Hence, high-level societal goals are translated into low-level technical requirements. Note the explicit difference between a societal goal and a measurable and precise security property.

Typically, there are a number of established techniques to verify a system satisfies a set of properties. One technique often used is manual inspection of the system's code, design, and procedures. This labor intensive process aims to either prove or disprove a specific property through reasoning. Doing so adequately requires reading and understanding the relevant parts of the system undergoing inspection. In a well designed system, it is possible to limit the scope of the system under consideration and study a smaller portion of it.

Another technique, called *static analysis*, involves using computer programs to analyze source code to validate security properties and is built upon a wealth of prior work. Static analysis tools attempt to automate the process of manual human inspection. Depending upon the sophistication of the static analysis tool used and the difficulty of the property being analyzed, static analysis can require additional help from the programmer. For example, a particular analysis may require the programmer to add annotations to the source code, or possibly to rewrite the code and thereby make it easier for the static analysis tool.

Static analysis and manual inspection each offer the benefit of detecting security problems while the system is being designed and are able to catch security errors before the voting system is deployed in the field. Naturally, there is a tremendous advantage to finding problems before any voter ever touches the voting machine; but for certain properties, it may be simpler to employ a *dynamic analysis*, whereby behavior that contradicts the property is detected while the voting machine is run, either during testing or in the course of an actual election. If the voting machine exhibits behavior that contradicts a security requirement, the DRE software can flag an error and prevent the voter from continuing. Dynamic analysis requires changes to the program code so it actively checks its own behavior. The programmer can enact the changes directly , or possibly with the assistance of a software tool.

This dissertation draws on all three techniques to prove a small set of properties, allowing us to gain confidence in certain aspects of a voting machine's behavior.

1.2 Contributions and summary of results

1.2.1 Properties

In Chapter 2 we outline high-level security goals for voting systems. These security goals are informed by convention, law, and social policy. As discussed, though, the security goals must be translated into more testable, concrete *properties* for voting systems. This chapter discusses six properties that we focus on during the course of this dissertation. We produce a voting system implementation in which we successfully verify three of the six properties and refer to additional work that details how to achieve similar success with the fourth property.

A fully verified voting machine would require verifying significantly more than a handful of properties. However, building and verifying all those properties in a voting machine is currently too daunting for us to consider. Recognizing that we should keep this as an end goal, we must start by verifying a few key properties. Current voting machines are not designed with verification in mind. Consequently, there is much value in a voting machine where it is possible to verify even a few properties. This is a positive first step.

1.2.2 Cryptographic voting protocols and privacy implications

Cryptographic voting protocols provide voters with a novel mechanism to verify their vote is properly recorded and counted. They are meant to augment DREs and provide voters with an end-to-end guarantee of the proper tabulation of their vote. Proponents of cryptographic voting protocols cite the end-to-end verifiability property as a reason for requiring less scrutiny of the software on these DREs. They argue that a vigilant voter would detect the effects of tampering by buggy software or malicious poll workers. This would lessen the necessity to trust the software since the voter provides an end-to-end check of their ballot’s integrity.

When using a cryptographic voting protocol, the voter typically takes home a receipt. For privacy protection, the receipt is specially designed to not reveal any of the voter’s choices. These protocols usually expect the users to check their receipt with an online version after voting; this check ensures the proper recording and counting of their vote. They can detect tampering or buggy voting machines via mathematical proofs of correctness.

Cryptographic voting protocols offer the promise of verifiable voting without the need to trust the integrity of any software in the system. However, these cryptographic protocols are only one part of a larger system composed of voting machines, software implementations, and election procedures, and we must analyze their security by considering the system in its entirety. In Chapter 3, we analyze the security properties of two different cryptographic protocols, one proposed by Andrew Neff and another by David Chaum. We discovered several potential weaknesses in these voting protocols which only became apparent when considered in the context of an entire voting system. These weaknesses include: subliminal channels in the encrypted ballots and denial of service attacks. These attacks could compromise election integrity, erode voter privacy, and enable vote coercion. Whether the attacks succeed or not will depend on how these ambiguities are resolved in a full implementation of a voting system, but we expect that a well designed implementation and deployment may be able to mitigate or even eliminate the impact of these weaknesses. However, these protocols must be analyzed in the context of a complete specification of the system and surrounding procedures before they are deployed in any large scale public election.

So, while the protocols offer the promise of skipping verification, their current implementations do not offer the same guarantees that the theoretical results would indicate. This gap in the realized systems means that as currently conceived, it is still necessary to verify security properties about the software implementation.

1.2.3 Privacy through reboots

The privacy problems present in cryptographic voting protocols are prevalent in other voting technologies as well. In Chapter 4, we cover privacy problems for a range of voting technologies. We introduce a simple idea to cut down on privacy leaks: rebooting after each voter. We outline the solution and then describe the conditions necessary to implement reboots to help alleviate privacy concerns. This technique, when combined with restrictions on how a program accesses its persistent storage, allows one to show that information from one voter's session cannot leak to another voter's session.

Employing this reboot technique to guarantee privacy need not be limited to voting applications. It is also of independent interest, and is likely applicable in other computation domains where users share the same hardware one after another in independent sessions. For example, users may demand privacy guarantees from the ATM machines or transit kiosks they use since they provide each machine with their financial details in conducting their transactions.

1.2.4 An architecture to verify voting

Realizing that we need new techniques to prove that specific security properties hold in voting machines, we explore a particular architecture specifically designed to make verification easier. In Chapter 5, we use specific properties about voting, off the shelf hardware, isolation, and architectural decisions to allow easy verification of two critical security properties.

We develop the architecture in a series of design exercises driven by two specific properties that we introduce in Chapter 2. We expand upon the privacy-reboot idea from Chapter 4 in a real system and implement it. The final design facilitates manual verification of these security properties, which we also discuss. Finally, we present the voting system's design and discuss our experience building a prototype implementation in Java and C.

1.2.5 Dynamically verifying properties

Some properties are best verified using software analysis. In Chapter 6, we look at proving the correctness of serialization—the process of storing the in-memory representation of a data structure, such as a ballot, to a permanent store such as a disk. Trusting computerized voting requires that serialization, and its mate deserialization, work together reliably and predictably.

We propose to use a dynamic check to guarantee proper recovery of the ballot from storage. Before the ballot is to be stored to disk, the DRE checks that the tallier (used to count the votes) will be able to be reconstruct the serialized ballot for proper counting at the end of the day. If an error is found, the voting machine alerts the voter and election officials of the error and refuses to proceed. Since the tallier is to be run later under potentially different conditions, the check must guarantee that deserialization will always yield the same results, even in a potentially different execution environment. For a deserialize function to always yield the same result, its return value must only depend on its arguments and any constants compiled into the code. It may not depend on non-deterministic inputs. We call such functions *environment-free*. We develop a static analysis to check the environment-free property in Java code. Proving the deserialize function is environment-free allows enables the DRE to check at run-time that the serialized ballot will always be able to properly to be deserialized. We describe the results of the environment free static checker and the results of using it to prove the correctness of serialization.

The *environment-free* checker is potentially useful to check other functions that follow the serialization/deserialization pattern. More broadly, serialization is just one of a family of common data transformation routines that litter programs. Two others in the family include encryption/decryption and compression/decompression. In Chapter 6, we show the checker also can be used to prove that decryption is the inverse of encryption for an AES implementation. We believe, therefore, that the environment-free checker is useful outside the voting context.

Chapter 2

Voting goals & properties

In this chapter, we start with an overview of the voting process. This will serve as useful background for the remaining chapters.

We then present a number of different security goals for voting systems. Goals reflect societal desires based on laws and convention. They make statements about the entire voting process, can often be subjective and stateable without many technical details. Goals guide system designers when they are forced to make engineering tradeoffs. The list of goals should not be seen as a static list; for example, the secret ballot, providing privacy and coercion resistance, was only adopted in the 1880s in the United States. The list of voting goals evolves with the advent of new technology. We consider six currently accepted goals, and one that may be on the horizon. Achieving these goals requires not only impeccable technology, but also stringent procedures, including voter education, machine maintenance, pollworker training, and dispute resolution. We concern ourselves with the behavior of the entire system, not just the voting technology.

But goals are not sufficient; it is still difficult to measure a voting system against a goal: a

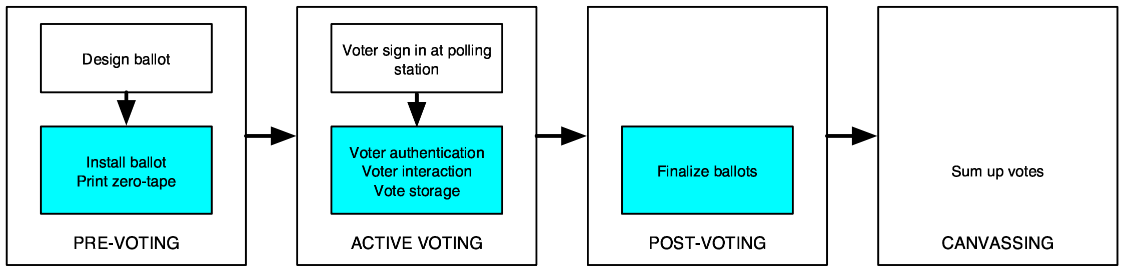


Figure 2.1: Major steps in the voting process when using DREs.

---

goal is broad and encompasses many facets. We must be very clear about what *specific* properties we aim to achieve in our system. A property is a more measurable requirement than a goal and is meant to be specific and objective; determining whether a voting system satisfies a property should not be ambiguous. An example property is that, when the voter is making their selection for a particular race, the voting system must present all candidates in a format in accordance with election laws. A voting machine that always exhibits the property could not conditionally omit certain candidates, or present certain candidates in a smaller font. Upon reading the source code, it should be possible to determine whether this property holds.

We focus on six properties that the rest of the dissertation addresses. The list is by no means exhaustive, but is chosen to reflect important properties that are first and important building blocks for any voting machine.

2.1 Voting overview

Pre-election setup. The full election process incorporates many activities beyond what a voter typically experiences in the voting booth. Although the exact processes differ depending on the specific voting technology in question, Figure 2.1 overviews the common steps for DRE-based voting. In the pre-election stage, election officials prepare ballot definition files describing the parameters of the election. Ballot definition files can be very complex [52], containing not only a list of races and information about how many selections a voter can make for each race, but also containing copies of the ballots in multiple languages, audio tracks for visually impaired voters (possibly also in multiple languages). Additionally, the ballot presented to the voter may vary based on the precinct as well as the voter's party affiliation. Election officials generally use external software to help them generate the ballot definition files. After creating the ballot definition files, an election worker will load those files onto the DRE voting machines. Before polls open, election officials generally print a "zero tape," which shows that no one cast a ballot prior to the start of the election.

Active voting. When voter Alice wishes to vote, she must first interact with election officials to prove that she is eligible to vote. The election officials then give her some token or mechanism to allow her to authenticate herself to the DRE as an authorized voter. Once the DRE verifies the token, the DRE displays the ballot information appropriate for Alice, e.g., the ballot might be in Alice's native language or, for primaries, be tailored to Alice's party affiliation. After Alice selects the candidates she wishes to vote for, the DRE displays a "confirmation screen" summarizing Alice's selections. Alice can then either accept the list and cast her ballot, or reject it and return to editing her selections. Once she approves her ballot, the DRE stores the votes onto durable storage and invalidates her token so that she cannot vote again.

Finalization & post-voting. When the polls are closed, the DRE ensures that no further votes can be cast and then prints a "summary tape," containing an unofficial tally of the number of votes for each candidate. Poll workers then transport the removable storage medium containing cast ballot images, along with the zero tape, summary tape, and other materials, to a central facility for tallying. During the canvass, election officials accumulate vote totals and cross-check the consistency of all these records.

Additional steps. In addition to the main steps above, election officials can employ various auditing and testing procedures to check for malicious behavior. For example, some jurisdictions use parallel testing, which involves sequestering a few machines, entering a known set of votes, and checking whether the final tally matches the expected tally. Also, one could envision repeating the vote-tallying process with a third-party tallying application, although we are unaware of any instance where this particular measure has been used in practice. While these additional steps can help detect problems, they are by no means sufficient.

2.2 Voting goals

In this section, we enumerate a number of broad goals for voting systems.

Goal 1. One voter/one vote: *The cast ballots should exactly represent the votes cast by legitimate voters. Malicious parties should not be able to add, duplicate, or delete ballots.*

This goal emphasizes that each legitimate voter should have exactly one vote toward each race. It should be impossible for the voters themselves, designers of the voting technology, election officials, or other people to subvert this goal. Procedures and voting policy can greatly impact whether this particular goal is successfully achieved. For example, it is imperative that the polling station be staffed with adequate supplies of voting materials (whether it be voting machines or blank ballots). Insufficient allocation or resources impinges on this goal; poor technology design can also adversely affect the goal, either by increasing the amount of resources needed, or through errors that can surreptitiously allow people to add or drop ballots at will. It also requires the poll workers to determine who is a legitimate voter.

Goal 2. Cast-as-intended: *A voter should be able to reliably and easily cast the ballot that they intend to cast.*

Cast-as-intended gets to the heart of voting – in essence, the voter must be able to reliably and consistently express their desired opinion for a particular election. Meeting this goal requires overcoming many challenges. Broadly, 1) the voting machine must present all choices for their particular ballot in a non-biased manner. As subtle changes in layout, order, or presentation can influence the voter to favor one choice over another, the voting machine must present all choices in as equitable manner as possible; 2) the voter must be able to express their desires among the choices. The voting machine should not make it more difficult to chose one candidate over another; 3) the completed ballot must be stored without changes and kept for tallying under all conditions. It is also imperative that the voter must be able to express their selections easily and efficiently and should strive to reduce inadvertent errors.

There are a host of issues underlying each of the three above challenges. As just one example, a voter who is unfamiliar with computers must have the same opportunities to express their votes as a computer-literate person. On electronic voting technology, this can be challenging. Designing user interfaces and ballot layouts that are unambiguous to first-time users is challenging.

Goal 3. Counted-as-cast: *The final tally should be an accurate count of the ballots that have been cast.*

The counted-as-cast goal assures the accuracy of the final tally. Achieving this goal requires that ballots are not modified or lost, and will properly be reconstructed in a form that reflects the original cast ballot form. The challenge is assuring this despite poor procedures, lost or broken voting machines, and ambiguities in determining the voter's intent.

Goal 4. Verifiability: *It should be possible for participants in the voting process to prove that the voting system obeys certain properties. For example, when referring to goals 2 and 3 (cast-as-intended and counted-as-cast), the voter should be able to prove to themselves that their ballot own ballot was cast-as-intended, and all voters should be able to prove to themselves (and others) that all of the ballots are properly counted-as-cast.*

Verifiability is a property that allows voting participants to easily prove the correct operation of some portion of the voting process. When discussing verifiability, it is critical to consider who is verifying the particular property under consideration. When the voter is performing the verification, it is imperative to consider the usability of the verification process. A property cannot reasonably called verifiable by the voter if it requires the voter to analyze source code. It would take the average voter far too long to learn the required skills. However, it would be appropriate to call such a property verifiable by software experts since they possess the required skills.

In this dissertation, we seek to enable software experts to verify a set of security properties.

Chapter 3 analyzes two cryptographic voting protocols that provide verifiably cast-as-intended and verifiably counted-as-cast to the voters. *Verifiably cast-as-intended* means each voter should be able to verify her ballot accurately represents the vote she cast. Often, this includes looking at a website after voting. *Verifiably counted-as-cast* means everyone should be able to verify that the final tally is an accurate count of the ballots contained on the website, for example. The difficult in achieving verifiability is doing so while also preserving a voter's privacy. Typically, solutions that strive for verifiability of cast-as-intended and counted-as-cast include at least some cryptographic techniques.

Goal 5. Privacy: *Ballots and all events during the voting process should be remain secret.*

A voter should be able to trust that their ballot and all interactions with the voting machine will remain hidden. In cases where the ballot is published, it should not be possible to link the ballot with the voter. The first part of the goal would even preclude indirect privacy leaks, whereby the voting machine changes its behavior in response to votes that have already been cast. Preserving privacy requires effort from the voting machine designers as well as the poll workers, since lapses by either can result in privacy leaks.

Goal 6. Coercion resistance: *A voter should not be able to prove how she voted to a third party not present in the voting booth.*

Coercion resistance is related to privacy. A voter should not be able to collude with an outsider in order to prove how they voted. Put another way, a voter should not be able to subvert their own privacy. There is a typical caveat with this goal: coercion resistance is not offered when the voter brings another person (or the electronic equivalent: a recording device) into the polling booth with them. In this case, the voter's companion can directly observe all of the voter's interactions with the voting machine

2.3 Specific properties

As stated, properties are measurable aspects of a voting system goals. One must be careful in which properties are required. It is possible that designing a voting system to exhibit one security property may help one goal to the detriment of another. As one example, a property requiring voting systems to provide voters with a printout of their onscreen selections to take home may help guarantee cast-as-intended, but at the cost of coercion resistance.

Resolving these tradeoffs requires guidance from policy makers. They are in the best position to guard and balance different stakeholders' interests. It is the job of computer scientists to point out the tradeoffs.

We now present specific properties that this dissertation work will address. These properties represent some aspect of one or more of the above goals, but aren't sufficient on their own to guarantee any of these goals are met.

Property 1. *None of a voter's interactions with the voting machine, including the final ballot, can affect any subsequent voter's sessions*[1].

This property has implications for Goals 2 and 5. A DRE that achieves Property 1 will prevent two large classes of attacks: one against election integrity and another against privacy. One way to understand this property is to consider a particular voting system design that exhibits the property.

[1] Note that some interactions may be unavoidable. For example, an electronic ballot box that becomes "full" on a voting machine should not allow subsequent voters to vote. This interaction is a desired and unavoidable interaction. The remedy here is to ensure that if the ballot box becomes full, there will be no subsequent voters.

A DRE can be "memoryless," so that after indelibly storing the ballot, it erases all traces of the voter's actions from its RAM. This way, a DRE cannot use the voter's choices in making future decisions.

A DRE that is memoryless cannot decide to change its behavior in the afternoon on election day if it sees the election trending unfavorably for one candidate. Similarly, successful verification of this property guarantees that a voter, possibly with the help of the DRE or election insider, cannot learn how a prior voter voted.

We discuss this property in Chapters 3 and 4.

Property 2. *A ballot cannot be cast without the voter's consent to cast it.*

Property 2 ensures the voter's ballot is only cast with their consent; a voting machine that always exhibits this property will help achieve Goal 2 (Cast-as-intended). When a ballot is cast with the voter's consent and at the proper time, guarantees that the voter has had the chance to see all races and has had the option of editing their selections before casting. Additionally, when combined with other security measures, this property helps guarantee the ballot box cannot be stuffed by the DRE. If each cast operation requires a human's input, and the DRE cannot automatically cast additional ballots.

Property 3. *The DRE cannot leak information through the on-disk format. Additionally, the ballot box should be history-independent and tamper evident.*

Part of Property 3 directly supports Goal 5 (Privacy). Requiring the on-disk format to be history-independent means that it should not leak the order that voters voted on the DRE. A DRE exhibiting this property would reduce the burden on procedures to safeguard the electronic ballot box. If the ballot box were not history-independent, the ballot box would contain the order in which voters voted. It would then be easy for an adversary to correlate the order in which voters voted with the order in which they entered the polling station and then link ballots to people. This ultimately compromises voter privacy.

This property can also further Goal 3 (Counted-as-cast). If the on-disk format of the ballot box does not reveal the vote order, it may be possible to publish an exact copy of the ballot box. This allows anyone to collate the ballot boxes from all DREs in a precinct and recreate the final tally to double check the tabulation process[2]. The ballot box must be history-independent in order to safely publish it.

We can use the techniques developed in conjunction with Molnar et al. in implementing Property 3 [55].

Property 4. *The DRE only stores ballots that have been approved by the voter.*

Property 4 refers to a few conditions. The DRE must not change the ballot after the voter chooses their candidates. Additionally, the voter must have a chance to see the contents of the ballot and approve or reject it. The ballot structure may be passed through confirmation screens and to serialization mechanisms before it is ultimately stored; through all this, it must remain unmodified. This is another aspect of Goal 2 (Cast-as-intended).

Property 5. *There should be a canonical format for the ballot so there is only one way to represent the voter's choices.*

Violation of Property 5 could violate the voter's privacy, even if the voter approves the ballot. Suppose the voter's choice, "James Polk" were stored with an extra space: "James␣␣Polk". The voter

[2]However, there are some subtleties to publishing the ballot boxes: if the votes are to be published, they must be done in a manner that does not enable vote-selling. For example, a vote-buyer may offer cash if a voter makes a selection for a high-profile race and then fills in a particular string for a write-in candidate in a different race. The vote-buyer will only pay if one ballot among the published ballots contains the pre-arranged string and a vote for the candidate they ordered.

would not likely notice anything were amiss, but this could convey privacy leaking-information in a subliminal channel, described in Chapter 3.

Property 6. *The ballot counted in the tally stage should be the same as the in-memory copy approved by the voter at the voting machine.*

This property, an aspect of Goal 3 (Counted-as-cast), guarantees that the ballot recording software can properly hand off the ballot to the tally machine. It requires that the serialized version of the in-memory ballot the voter fills out must be properly deserialized into an equivalent in-memory copy when needed by the tallying software.

We do not expect these to be an exhaustive list of the desirable security properties; rather, they are properties that we believe are important and that we can easily achieve with the contributions of this work.

Chapter 3

Cryptographic voting protocols

In this chapter, we look at two cryptographic voting protocols. They provide the voter the opportunity to verify their own vote was cast-as-intended and that all votes were counted-as-cast. This is a major step forward in the capabilities of voting systems.

However, in this chapter, we show it is imperative to view cryptographic protocols as a part of a complete voting system and consider the security implications of all surrounding procedures and the implementations of the protocols. Doing so for these protocols reveals privacy vulnerabilities through subliminal channels (the ramifications of which will be mitigated through some strategies suggested in Chapter 4), and opportunities for denial of service attacks.

Parts of this work are drawn with permission from previously published work [39].

3.1 Introduction

Trustworthy voting systems are crucial for the democratic process. Recently, *direct recording electronic* voting machines (DREs) have come under fire for failing to meet this standard. The

Weakness	**Protocols**	**Threat Model**	**Affects**
Random subliminal channels	Neff	Malicious DRE colluding w/ outsider	Voter privacy & coercion resistance
Semantic subliminal channels	Chaum	Malicious DRE colluding w/ outsider	Voter privacy & coercion resistance
Denial of service attacks	Neff & Chaum	Malicious DRE or tallying software	Voter confidence & election integrity

Table 3.1: Summary of weaknesses we found in Neff's and Chaum's voting schemes.

problem with paperless DREs is that the voting public has no good way to tell whether votes were recorded or counted correctly, and many experts have argued that, without other defenses, these systems are not trustworthy [42, 57].

Andrew Neff and David Chaum have recently proposed revolutionary schemes for DRE-based electronic voting [19, 60, 61]. The centerpiece of these schemes consists of novel and sophisticated cryptographic protocols that allow voters to verify their votes are cast and counted correctly. Voting companies Votegrity and VoteHere have implemented Chaum's and Neff's schemes, respectively. These schemes represent a significant advance over previous DRE-based voting systems: voters can verify that their votes have been accurately recorded, and everyone can verify that the tallying procedure is correct, preserving privacy and coercion resistance in the process. The ability for anyone to verify that votes are counted correctly is particularly exciting, as no prior system has offered this feature.

This chapter presents a first step towards a security analysis of these schemes. Our goal is to determine whether these new DRE-based cryptographic voting systems are trustworthy for use in public elections. We approach this question from a systems perspective. Neff's and Chaum's schemes consist of the composition of many different cryptographic and security subsystems. Composing security mechanisms is not simple, since it can lead to subtle new vulnerabilities [28, 48, 64].

Consequently, it is not enough to simply analyze a protocol or subsystem in isolation, as some attacks only become apparent when looking at an entire system. Instead, we perform a whole-system security analysis.

In our analysis of these cryptographic schemes, we found weaknesses in that subliminal channels may be present in the encrypted ballots. These attacks could potentially compromise election integrity, erode voter privacy, and enable vote coercion. In addition, we found several detectable but unrecoverable denial of service attacks. We note that these weaknesses only became apparent when examining the system as a whole, underlining the importance of a security analysis that looks at cryptographic protocols in their larger systems context.

The true severity of the weaknesses depends on how these schemes are finally implemented. During our security analysis, one challenge we had to deal with was the lack of a complete system to analyze. Although Neff and Chaum present fully specified cryptographic protocols, many implementation details—such as human interfaces, systems design, and election procedures—are not available for analysis. Given the underspecification, it is impossible to predict with any confidence what the practical impact of these weaknesses may be. Consequently, we are not yet ready to endorse these systems for widespread use in public elections. Still, we expect that it may be possible to mitigate some of these risks with procedural or technical defenses, and we present countermeasures for some of the weaknesses we found and identify some areas where further research is needed. Our results are summarized in Table 3.1.

3.2 Preliminaries

David Chaum and Andrew Neff have each proposed a cryptographic voting protocol for use in DRE machines [13, 19, 60, 61, 89]. Although these protocols differ in the details of their operation, they are structurally similar. Both protocols fit within the DRE voting steps in Figure 2.1. However, they introduce a few extra actions, which we outline here.

In the pre-voting stage, a set of *election trustees* with competing interests are chosen such that it is unlikely that all trustees will collude. The trustees interact amongst themselves before the election to choose parameters and produce key material used throughout the protocol. The trustees should represent a broad set of interest groups and governmental agencies to guarantee sufficient separation of privilege and discourage collusion among the trustees.

Active voting begins when a voter visits a polling station to cast her vote on election day, and ends when that ballot is cast. To cast her vote, the voter interacts with a DRE machine in a private voting booth to select her ballot choices. The DRE then produces an electronic ballot representing the voter's choices and posts this to a public bulletin board. This public bulletin board serves as the ballot box. At the same time, the DRE interacts with the voter to provide a *receipt*. Receipts are designed to resist vote buying and coercion, and do not allow the voter to prove to a third party how she voted. Also, each voter's ballot is assigned a unique *ballot sequence number* (BSN). BSNs ease auditing and verification procedures, without compromising voter privacy.

After all ballots have been posted to the bulletin board, canvassing stage begins. The election trustees execute a publicly verifiable multistage mix net, where each trustee privately executes a particular stage of the mix net [33, 61]. To maintain anonymity, the trustees strip each ballot of its BSN before it enters the mix net. Each stage of the mix net takes as input a set of encrypted ballots, partially decrypts or re-encrypts them (depending on the style of mix net), and randomly permutes them. The final result of the mix net is a set of plaintext ballots which can be publicly counted but which cannot be linked to the encrypted ballots or to voter identities. In cryptographic voting protocols, the mix net is designed to be universally verifiable: the trustee provides a proof which any observer can use to confirm that the protocol has been followed correctly. This means a corrupt trustee cannot surreptitiously add, delete, or alter ballots.

At various points during this process, voters and observers may engage in election verification. After her ballot has been recorded on the public bulletin board, the voter may use her receipt to verify her vote was cast as intended and will be accurately represented in the election results. Note that the receipt does not serve as an official record of the voter's selections; it is only intended for convincing the voter that her ballot was cast correctly. Election observers (e.g., the League of Women Voters) can verify certain properties about ballots on the public bulletin board, such as, that all ballots are well-formed or that the mix net procedure was performed correctly.

Both the Chaum and Neff protocols require DREs to contain special printing devices for providing receipts. The security requirements for the printer are: 1) the voter can inspect its output, and 2) neither the DRE nor the printer can erase, change, or overwrite anything already printed without the voter immediately detecting it. There are some differences in the tasks these devices perform and additional security requirements they must meet, which we will discuss later.

3.2.1 Threat models

We must consider a strong threat model for voting protocols. In national elections, billions of dollars are at stake, and even in local elections, controlling the appropriation of municipal funding in a large city can be sufficient motivation to compromise significant portions of the election system [41]. We consider threats from three separate sources: DREs, talliers, and outside coercive parties. To make matters worse, malicious parties might collude together. For example, malicious DREs might collude with outside coercers to buy votes.

Malicious DREs can take many forms [5]. A programmer at the manufacturer could insert Trojan code, or a night janitor at the polling station could install malicious code the night before the election. We must assume malicious DREs behave arbitrarily. Verification of all the DRE software in an election is hard, and one goal of Neff's and Chaum's schemes is to eliminate the need to verify that the DRE software is free from Trojan horses.

We also must consider malicious parties in the tallying process, such as a malicious bulletin board or malicious trustees. These parties wield significant power, and can cause large problems if they are malicious. For example, if the bulletin board is malicious, it can erase all the ballots. If all the software used by the trustees is malicious, it could erase the private portions of the trustees' keys, making ballot decryption impossible.

To evaluate a voting system's coercion resistance, we must consider outside coercive parties colluding with malicious voters. We assume the coercer is not present in the voting booth. Attacks where the coercer is physically present are outside the scope of voting protocols and can only be countered with physical security mechanisms. Similarly, attacks where a voter records her actions in the poll booth (e.g., with a video or cell phone camera) are also outside the scope of voting protocols, and we do not consider them here.

Finally, we must consider honest but unreliable participants. For example, voters and poll workers might not fully understand the voting technology or utilize its verification properties, and a malicious party might be able to take advantage of this ignorance, apathy, or fallibility to affect the outcome of the election.

3.3 Two voting protocols

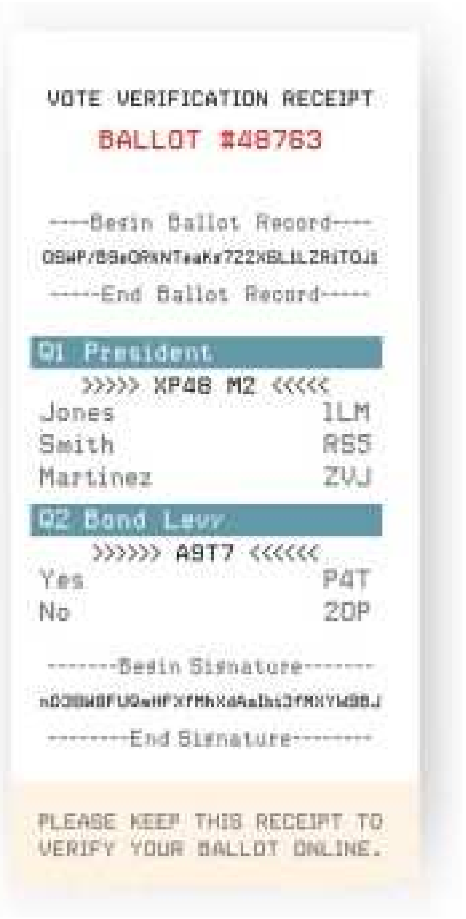
In this section, we describe Neff's and Chaum's voting protocols in detail.

3.3.1 Neff's scheme

Andrew Neff has proposed a publicly verifiable cryptographic voting protocol for use in DREs [60, 61]. During election initialization, the trustees perform a distributed key generation protocol to compute a master public key; decryption will only be possible through the cooperation of all trustees in a threshold decryption operation. Also, there is a security parameter ℓ. A DRE can surreptitiously cheat with a probability of $2^{-\ell}$. Neff suggests $10 \leq \ell \leq 15$.

Neff's scheme is easily extensible to elections with multiple races, but for the sake of simplicity assume there is a single race with candidates $C_1, \ldots, C_n$. After a voter communicates her choice C_i to the DRE, the DRE constructs an encrypted electronic ballot representing her choice and commits to it. Each ballot is assigned a unique BSN. The voter is then given the option of interacting with the DRE further to obtain a receipt. In Figure 3.1, we show an example of a receipt taken from the VoteHere website. This receipt enables the voter to verify with high probability that her vote is accurately represented in the tallying process.

After the voter communicates her intended choice C_i to the DRE, it constructs a *verifiable choice* (VC) for C_i. A VC is essentially an encrypted electronic ballot representing the voter's choice C_i (see Figure 3.2). A VC is a $n \times \ell$ matrix of *ballot mark pairs* (BMPs), one row per candidate (recall that ℓ is a security parameter). Each BMP is a pair of El Gamal ciphertexts. Each



```
VOTE VERIFICATION RECEIPT
BALLOT #48763

----Begin Ballot Record----
OBWP/B9sORNNTeaKs722XBLlLZRiTOJi
-----End Ballot Record-----

Q1 President
>>>>> XP48 M2 <<<<<
Jones                  1LM
Smith                  RS5
Martinez               ZVJ
Q2 Bond Levy
>>>>>> A9T7 <<<<<<
Yes                    P4T
No                     2DP

-------Begin Signature-------
nO3BW8FUQwHFXfMhXdAaIbi3fMXYWBBJ
--------End Signature--------

PLEASE KEEP THIS RECEIPT TO
VERIFY YOUR BALLOT ONLINE.
```

Figure 3.1: This is an example of a detailed receipt for Neff’s scheme, taken from the VoteHere website, `http://www.votehere.com`.

$$\begin{array}{cccccc} & 1 & 2 & 3 & & \ell \\ C_1 & \boxed{0}\,\boxed{1} & \boxed{1}\,\boxed{0} & \boxed{0}\,\boxed{1} & \cdots & \boxed{0}\,\boxed{1} \\ C_2 & \boxed{1}\,\boxed{1} & \boxed{1}\,\boxed{1} & \boxed{0}\,\boxed{0} & \cdots & \boxed{1}\,\boxed{1} \\ C_3 & \boxed{1}\,\boxed{0} & \boxed{0}\,\boxed{1} & \boxed{0}\,\boxed{1} & \cdots & \boxed{1}\,\boxed{0} \\ & \vdots & \vdots & \vdots & & \vdots \\ C_n & \boxed{1}\,\boxed{0} & \boxed{1}\,\boxed{0} & \boxed{0}\,\boxed{1} & \cdots & \boxed{1}\,\boxed{0} \end{array}$$

Figure 3.2: A verifiable choice (VC) in Neff's scheme. $\boxed{b}$ represents an encryption of bit b. This VC represents a choice of candidate C_2. Note the second row contains encryptions of $(0,0)$ and $(1,1)$, and the unchosen rows contain encryptions of $(0,1)$ and $(1,0)$.

ciphertext is an encryption of 0 or 1 under the trustees' joint public key, written $\boxed{0}$ or $\boxed{1}$ for short. Thus, each BMP is a pair $\boxed{b_1}\,\boxed{b_2}$, an encryption of (b_1, b_2).

The format of the plaintexts in the BMPs differs between the row corresponding to the chosen candidate C_i (i.e., row i) and the other ("unchosen") rows. Every BMP in row i should take the form $\boxed{0}\,\boxed{0}$ or $\boxed{1}\,\boxed{1}$. In contrast, the BMPs in the unchosen rows should be of the form $\boxed{0}\,\boxed{1}$ or $\boxed{1}\,\boxed{0}$. Any other configuration is an indication of a cheating or malfunctioning DRE. More precisely, there is a $n \times \ell$ matrix x so that the k-th BMP in unchosen row j is $\boxed{x_{j,k}}\,\boxed{\sim x_{j,k}}$, and the k-th BMP in the choice row i is $\boxed{x_{i,k}}\,\boxed{x_{i,k}}$.

Consider the idealized scenario where all DREs are honest. The trustees can tally the votes by decrypting each ballot and looking for the one row consisting of $(0,0)$ and $(1,1)$ plaintexts. If decrypted row i consists of $(0,0)$ and $(1,1)$ pairs, then the trustees count the ballot as a vote for candidate C_i.[1]

In the real world, we must consider cheating DREs. Up to this point in the protocol, the DRE has constructed a VC supposedly representing the voter's choice C_i, but the voter has no assurance this VC accurately represents her vote. How can we detect a dishonest DRE?

[1]This is a simplified view of how the trustees tally votes in Neff's scheme, but it captures the main idea.

Neff's scheme prints the pair $(\mathrm{BSN}, \mathrm{hash}(VC))$ on the receipt and then splits verification into two parts: 1) at the polling booth, the DRE will provide an interactive proof of correct construction of the VC to the voter; 2) later, the voter can compare her receipt to what is posted on the bulletin board to verify that her ballot will be properly counted. At a minimum, this interactive protocol should convince the voter that row i (corresponding to her intended selection) does indeed contain a set of BMPs that will be interpreted during tallying as a vote for C_i, or in other words, each BMP in her chosen row is of the form $\boxed{b}\,\boxed{b}$. Neff introduces a simple protocol for this: for each such BMP, the DRE provides a *pledge* bit p; then the voter randomly selects the left or right position and asks the DRE to provide a proof that the ciphertext in that position indeed decrypts to p; and the DRE does so by revealing the randomness used in the encryption. Here we are viewing the ciphertext $\boxed{b}$ as a commitment to b, and $\boxed{b}$ is *opened* by revealing b along with the randomness used during encryption. If this BMP has been correctly formed as $\boxed{b}\,\boxed{b}$, the DRE can always convince the voter by using the value b as a pledge; however, if the BMP contains either $\boxed{0}\,\boxed{1}$ or $\boxed{1}\,\boxed{0}$, the voter has a $\frac{1}{2}$ probability of detecting this. By repeating the protocol for each of the ℓ BMPs in row i, the probability that a malformed row escapes detection is reduced to $(\frac{1}{2})^{\ell}$. The role of the interactive protocol is to ensure that the receipt will be convincing for the person who was in the voting booth but useless to anyone else.

In practice, it is unrealistic to assume the average voter will be able to parse the VC and carry out this protocol unassisted within the polling station. Instead, Neff's scheme enables the voter to execute it later with the assistance of a trusted software program. The DRE first prints the pledges on the receipt, and then receives and prints the voter's challenge. The challenge c_i for the row i is represented as a bit string where the k-th bit equal to 0 means open the left element of the

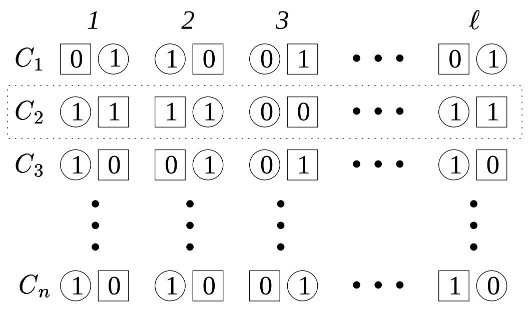


Figure 3.3: An opened verifiable choice (OVC) in Neff's scheme. $\boxed{b}$ represents an encryption of bit b, and $\textcircled{b}$ represents an opened encryption of bit b. An opened encryption of b contains both b and the randomness ω used to encrypt b in the VC.

k-th BMP and 1 means open the right element.

The DRE then constructs an *opened verifiable choice* (OVC) according to the voter's challenge and submits it to the bulletin board. In Figure 3.3, we show an example of an OVC constructed from the VC in Figure 3.2. We represent an opened encryption of bit b in an half-opened BMP by $\textcircled{b}$. In the OVC, the opened BMPs in row i are opened according to c_i, so that each half-opened BMP contains a pair of the form $\textcircled{b}\,\boxed{b'}$ (if $c_{i,k} = 0$) or $\boxed{b}\,\textcircled{b'}$ (if $c_{i,k} = 1$). To ensure that the OVC does not reveal which candidate was selected, the BMPs in the unchosen rows are also half-opened. In unchosen row j, the DRE selects an ℓ-bit challenge c_j uniformly at random and then opens this row according to c_j. Thus, an OVC consists of an $n \times \ell$ matrix of half-opened

BMPs. Consequently, the usual invocation of the receipt formation protocol is as follows:

$$\begin{array}{ll}
1.\ \text{Voter} \to \text{DRE}: & i \\
2.\ \text{DRE} \to \text{Printer}: & \text{BSN}, \text{hash}(VC) \\
3.\ \text{DRE} \to \text{Printer}: & \text{commit}(p_1, \ldots, p_n) \\
4.\ \text{Voter} \to \text{DRE}: & c_i \\
5.\ \text{DRE} \to \text{Printer}: & c_1, \ldots, c_n \\
6.\ \text{DRE} \to \text{B. Board}: & OVC
\end{array}$$

Here we define $p_{i,k} = x_{i,k}$ and $p_{j,k} = x_{j,k} \oplus c_{j,k}$ ($j \neq i$). While at the voting booth, the voter only has to check that the challenge c_i she specified does indeed appear on the printed receipt in the i-th position (i.e., next to the name of her selected candidate). Later, the voter can check that the OVC printed in step 5 does appear on the bulletin board and matches the hash printed in step 2 (and that the candidates' names are printed in the correct order), and that the OVC contains valid openings of all the values pledged to in step 3 in the locations indicated by the challenges printed in step 5. Note that the VC can be reconstructed from the OVC, so there is no need to print the VC on the receipt or to post it on the bulletin board.

To prevent vote buying and coercion, the voter is optionally allowed to specify challenges for the unchosen rows between steps 2 and 3, overriding the DRE's default random selection of c_j ($j \neq i$). If this were omitted, a vote buyer could tell the voter in advance to vote for candidate C_i and to use some fixed value for the challenge c_i, and the voter could later prove how she voted by presenting a receipt with this prespecified value appearing as the i-th challenge.

After the election is closed, the trustees apply a universally verifiable mix net to the collection of posted ballots. Neff has designed a mix net for El Gamal pairs [58, 61], and it is used here.

1.	Voter $\rightarrow$ DRE :	i
2.	DRE $\rightarrow$ Printer :	$\text{BSN}, \text{hash}(VC)$
3.	DRE $\rightarrow$ Voter :	basic or detailed?
4.	Voter $\rightarrow$ DRE :	r, where $r \in \{\text{basic}, \text{detailed}\}$
$5a$.	DRE $\rightarrow$ Printer :	$\text{commit}(p_1, \ldots, p_n)$
$5b$.	Voter $\rightarrow$ DRE :	c_i
$5c$.	DRE $\rightarrow$ Printer :	$c_1, \ldots, c_n$
6.	DRE $\rightarrow$ B. Board :	OVC

Figure 3.4: Summary of receipt generation in Neff's scheme with the option of basic or detailed receipts. Steps $5a, 5b$, and $5c$ happen only if $r =$ detailed.

In VoteHere's implementation of Neff's scheme, voters are given the option of taking either a *detailed* or *basic* receipt. The detailed receipt contains all the information described in this section (Figure 3.1), but a basic receipt contains only the pair (BSN, hash(VC)). This decision is made separately for each race on a ballot, and for each race that a voter selects a detailed receipt she must independently choose the choice and unchosen challenges for that race.

A basic receipt affords a voter only limited verification capabilities. Since a basic receipt foregoes the pledge/challenge stage of Neff's scheme, a voter cannot verify her ballot was recorded accurately. However, a basic receipt does have some value. It enables the voter to verify that the ballot the DRE committed to in the poll booth is the same one that appears on the bulletin board. Since the DRE must commit to the VC before it knows whether the voter wants a detailed or basic receipt, a DRE committing a VC that does not accurately represent the voter's selection is risking detection if the voter chooses a detailed receipt. The receipt protocol augmented with this additional

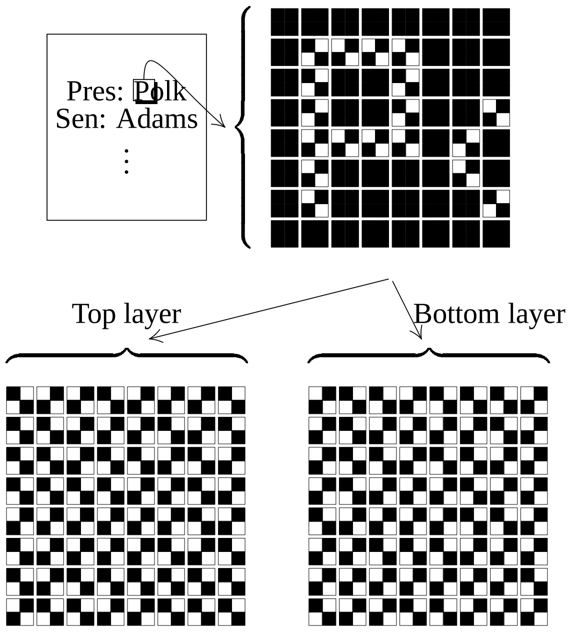


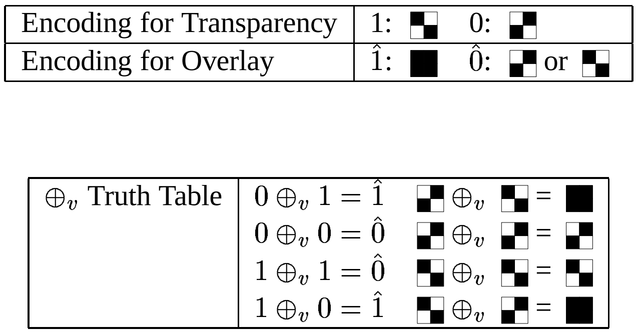
Figure 3.5: Representation of the printed ballot and transparencies in Chaum's scheme. The top two images show the ballot as well as a zoomed in portion of the two overlayed transparencies portrayed below.

choice is summarized in Figure 3.4.

3.3.2 Chaum's visual crypto scheme

David Chaum uses a two-layer receipt based on transparent sheets for his verifiable voting scheme [13, 19, 89]. A voter interacts with a DRE machine to generate a ballot image $\mathcal{B}$ that represents the voter's choices. The DRE then prints a special image on each transparency layer. The ballot bitmaps are constructed so that overlaying the top and bottom transparencies (T and B) reveals the voter's original ballot image. On its own, however, each layer is indistinguishable from a random dot image and therefore reveals nothing about the voter's choices (see Figure 3.5).

The DRE prints cryptographic material on each layer so that the trustees can recover the original ballot image during the tabulation phase. The voter selects either the top or bottom layer,



Encoding for Transparency	1: ▚ 0: ▞
Encoding for Overlay	$\hat{1}$: ■ $\hat{0}$: ▞ or ▚

$\oplus_v$ Truth Table	$0 \oplus_v 1 = \hat{1}$	▞ $\oplus_v$ ▚ = ■
	$0 \oplus_v 0 = \hat{0}$	▞ $\oplus_v$ ▞ = ▞
	$1 \oplus_v 1 = \hat{0}$	▚ $\oplus_v$ ▚ = ▚
	$1 \oplus_v 0 = \hat{1}$	▚ $\oplus_v$ ▞ = ■

Figure 3.6: Visual cryptography overview. A printed pixel on a single transparency has a value in $\{0, 1\}$, encoded as shown in the first row. We apply the visual xor operator $\oplus_v$ by stacking two transparencies so that light can shine through areas where the subpixels are clear. The pixels in the overlay take values from $\{\hat{0}, \hat{1}\}$. The bottom table shows the truth table for the visual xor operator and its parallels to the binary xor operator.

and keeps it as her receipt. A copy of the retained layer is posted on the bulletin board, and the other layer is destroyed. The voter can later verify the integrity of their receipt by checking that it appears on the bulletin board and that the cryptographic material is well formed.

Visual cryptography exploits the physical properties of transparencies to allow humans to compute the xor of two quantities without relying on untrusted software. Each transparency is composed of a uniform grid of *pixels*. Pixels are square and take values in $\{0, 1\}$. We print ▞ for a 0-valued pixel and ▚ for a 1-valued pixel. We refer to each of the four smaller squares within a pixel as *subpixels*. Overlaying two transparencies allows light to shine through only in locations where both subpixels are clear, and the above encoding exploits this so that overlaying performs a sort of xor operation. Pixels in the overlay take values in $\{\hat{0}, \hat{1}\}$. Pixels in the overlay have a different appearance than those in the individual transparency layer: $\hat{0}$ appears as ▞ or ▚, while $\hat{1}$ appears as ■. Using $\oplus_v$ to represent the visual overlay operation, we see that $0 \oplus_v 0 = \hat{0}$, $0 \oplus_v 1 = \hat{1}$, and in general if $a \oplus b = c$ then $a \oplus_v b = \hat{c}$ (see Figure 3.6).

Chaum's protocol satisfies three properties:

1. **Visual Check:** Given the desired ballot image $\mathcal{B}$, the DRE must produce two transparencies T and B so that $T \oplus_v B = \mathcal{B}$. This property allows the voter to verify the correct formation of the two transparencies.

2. **Recovery:** Given a single transparency T or B and the trustee keys, it must be possible to recover the original ballot image $\mathcal{B}$.

3. **Integrity:** T and B contain a commitment. There is a way to open T or B and to verify the opening so that for all other top and bottom pairs T' and B' such that $T' \oplus_v B' \approx \mathcal{B}$ and T' (or B') does not decrypt to $\mathcal{B}$, then B' (or T') is unopenable. In other words, for a pair of transparencies that overlay to form $\mathcal{B}$ (or a close enough approximation for the voter to accept it as $\mathcal{B}$), the DRE should only be able to generate a witness for a transparency if the other transparency decrypts to $\mathcal{B}$.

We will consider each pixel to have a type $\in \{\boxed{P}, \boxed{E}\}$ in addition to its value $\in \{0, 1\}$. The pixel's type will determine how we compute the value. We label pixels on the transparency so that no pixels of the same type are adjacent to each other, forming a repeating grid of alternating pixel types. Additionally, when the two transparencies are stacked, we require that $\boxed{P}$-pixels are only atop $\boxed{E}$-pixels and $\boxed{E}$-pixels are only atop $\boxed{P}$-pixels. The upper left corner of the top transparency looks like: $\begin{smallmatrix}\boxed{E}\boxed{P}\boxed{E}\\\boxed{P}\boxed{E}\boxed{P}\\\boxed{E}\boxed{P}\boxed{E}\end{smallmatrix}$, and the upper left corner of the bottom transparency looks like: $\begin{smallmatrix}\boxed{P}\boxed{E}\boxed{P}\\\boxed{E}\boxed{P}\boxed{E}\\\boxed{P}\boxed{E}\boxed{P}\end{smallmatrix}$. The $\boxed{P}$-pixels in a layer come from a pseudorandom stream. The stream is composed of n separate streams, one from each trustee. Each of these trustee streams is based on the trustee number and the voter's BSN; the seed will be encrypted using each trustee's public key requiring the trustee to participate in the decryption process. The value of the $\boxed{E}$-pixel is set so that overlaying it with the corresponding $\boxed{P}$-pixel in the other layer yields a ballot pixel. An $\boxed{E}$-pixel alone reveals no information: it is the xor of a $\boxed{P}$-pixel and the ballot image.

Details on transparency formation

The pseudorandom stream for a given transparency is composed of n pseudorandom streams, each of which is seeded by a different value. For each of the top and bottom transparencies, there is one stream per trustee. The i^{th} trustee's seed for the top is

$$st_i \triangleq h(\text{sign}_{k_t}(\text{BSN}), i) \tag{3.1}$$

where BSNrepresents the unique ballot sequence number assigned to the voter and $\text{sign}_{k_t}(\cdot)$ is a signature using k_t, a key specific to the DRE, and $h(\cdot)$ is a hash function. The i^{th} trustee's seed for the bottom is

$$sb_i \triangleq h(\text{sign}_{k_b}(\text{BSN}), i) \tag{3.2}$$

The hash expansion function $h'(\cdot)$ is used to generate the trustee stream. Trustee streams are xored together to produce the pseudorandom stream for the top layer:

$$PT \triangleq \bigoplus_{i=1}^{n} h'(st_i) \tag{3.3}$$

The corresponding bottom stream uses the bottom seeds:

$$PB \triangleq \bigoplus_{i=1}^{n} h'(sb_i) \tag{3.4}$$

We can now define each pixel's value. We view the ballot as a stream of pixels $\mathcal{B}$, and $\mathcal{B}[i]$ denotes the i^{th} pixel. A $\boxed{P}$-pixel i on the top transparency is assigned the value $PT[i]$. The $\boxed{E}$-pixel i on the bottom transparency is defined to have value $PT[i] \oplus \mathcal{B}[i]$. When viewing the two transparencies in alignment, then, the voter sees the original ballot stream $\mathcal{B}$ because $PT[i] \oplus_v$

$(PT[i] \oplus \mathcal{B}[i]) = PT[i] \oplus (PT[i] \oplus \mathcal{B}[i]) = \mathcal{B}[i]$. When taken alone, neither transparency reveals any information since each pixel is either pseudorandomly generated or the xor of a pseudorandom quantity and the original ballot.

After constructing the two layers, the DRE appends an onion encryption of the seeds so the trustees can jointly recover PT or PB. The DRE adds

$$\begin{aligned} DB &\triangleq e_{k_n}(sb_n || e_{k_{n-1}}(\ldots || e_{k_2}(sb_2 || e_{k_1}(sb_1)))) \\ DT &\triangleq e_{k_n}(st_n || e_{k_{n-1}}(\ldots || e_{k_2}(st_2 || e_{k_1}(st_1)))) \end{aligned} \tag{3.5}$$

to each transparency. DT and DB are known as dolls. $e_{k_i}(\cdot)$ is a public-key encryption function that uses the i^{th} trustee's public key, k_i.

The voter is then presented a choice to either choose the top or bottom transparency as a receipt. After the voter chooses a receipt layer, the DRE appends signatures committing to the voter's and its choices. Without loss of generality, assume the voter keeps the top transparency as a receipt. The DRE then prints $\text{sign}_{k_t}(\text{BSN})$ as an opening for the top layer (see the integrity requirement of the previous section). This opening allows the voter to verify that the DRE properly formed st_i and that the DRE printed the $\boxed{P}$-pixels on the chosen layer as it should. By recreating the onion encryption, the voter can verify that DT is properly formed. Finally, the DRE appends a copy of the chosen layer to the bulletin board. We show a summary of Chaum's protocol in Figure 3.7.

When the voter performs these checks, a malicious DRE has only a $1/2$ chance of evading detection. By extension, its chance of changing a significant number of ballots without being caught is exponentially small. For instance, a DRE can cheat by forming the $\boxed{P}$-pixels incorrectly so the voter will see what they expect in the overlay yet the ballot will decrypt to some other im-

$$\begin{array}{ll}
1.\ \text{Voter} \rightarrow \text{DRE}: & \text{candidate choices} \\
2.\ \text{DRE} \rightarrow \text{Printer}: & \text{transparency images} \\
3.\ \text{DRE} \rightarrow \text{Printer}: & \text{BSN}, DB, DT \\
4.\ \text{Voter} \rightarrow \text{Printer}: & c \qquad \text{where } c \in \{\text{top}, \text{bottom}\} \\
5.\ \text{DRE} \rightarrow \text{Printer}: & \text{sign}_{k_c}(\text{BSN}), \\
& \text{sign}_{k_{\text{DRE}}}(\text{BSN}, DT, DB, \text{chosen transparency})
\end{array}$$

Figure 3.7: Summary of Chaum's protocol.

age. However, the voter will detect cheating if her receipt transparency contains incorrectly formed $\boxed{P}$-pixels. Therefore, a malicious DRE must commit to cheating on either the top or bottom transparency (not both, or else it will surely be caught) and hope the voter does not choose that layer as a receipt.

Tabulation & verification

Chaum uses a Jakobsson et al. style mix net to decode the transparency chosen by the voter and recover their choices from $\mathcal{B}$ in the tallying phase [33]. The values of the pseudorandom pixels do not contain any information, while the encrypted pixels contain the ballot image xor-ed with the pseudorandom pixels from the other transparency. For each ballot that a trustee in the mix net receives, trustee i in the mix net recovers its portion of the pseudorandom stream. Let's assume the voter chose a top transparency. In the case, trustee i will first decrypt the doll provided by the DRE (Equation (3.5)) to obtain sb_i and then xor $h'(sb_i)$ into the $\boxed{E}$-pixels in the encrypted ballot. This trustee next permutes all of the modified ballots and passes the collection to the next trustee.

When the ballots exit the mix net, the $\boxed{P}$-pixels still contain pseudorandom data, but the encrypted pixels will contain the voter's ballot pixels from $\mathcal{B}$.

3.4 Subliminal channels

Subliminal channels, also known as covert communication channels, arise in electronic ballots when there are multiple valid representations of a voter's choices. If the DRE can choose which representation to submit to the bulletin board, then the choice of the representation can serve as a subliminal channel. Subliminal channels are particularly powerful because of the use of public bulletin boards in voting protocols. A subliminal channel in ballots on the bulletin board could be read by anyone (if the decoding algorithm is public) or only by a select few (if the decoding algorithm is secret).

A subliminal channel in an encrypted ballot carrying the voter's choices and identifying information about the voter threatens voter privacy and enables vote coercion. For example, as Keller et al. note, a DRE could embed in each encrypted ballot the time when the ballot was cast and who the voter chose for president [40]. Then, a malicious observer present in the polling place could record when each person voted and later correlate that with the data stored in the subliminal channel to recover each person's vote. Alternatively, if a malicious poll worker learns a voter's BSN, she can learn how a person voted since each encrypted ballot includes the BSN in plaintext. Detecting such attacks can be quite difficult: without specific knowledge of how to decode the subliminal channel, the encrypted ballots may look completely normal. The difficulty of detection, combined with the enormous number of voters who could be affected by such an attack, makes the subliminal channel threat troubling.

The above scenarios illustrate how an adversary can authentically learn how someone voted. Coercion then becomes simple: the coercer requires the voter to reveal their BSN or the time at which they voted, then later verifies whether there exists a ballot with that identifying information and the desired votes.

The threat model we consider for subliminal channel attacks is a malicious DRE colluding with an external party. For example, a malicious programmer could introduce Trojan code into DREs and then sell instructions on how to access the subliminal channel to a coercer.

Neither Neff's nor Chaum's protocol completely address subliminal channels in ballots. In this section, we present subliminal channel vulnerabilities in these protocols and some possible mitigation strategies.

One interesting observation is that subliminal channels are a new problem created by these protocols. Subliminal channels only become a serious problem because the bulletin board's contents are published for all to see. Since all the ballots are public and anonymously accessible, decoding the channel does not require any special access to the ballots. Subliminal channels are not a significant problem with current non-cryptographic DREs because electronic ballots are not public.

3.4.1 Randomness

Several cryptographic primitives in Neff's scheme require random values, and subliminal channel vulnerabilities arise if a malicious DRE is free to choose these random values.[2] These prim-

[2]Chaum's scheme, as originally published, does not specify which encryption primitives should be used to construct the onion encryption in Equation 3.5 [19]. Subsequently, Chaum has related to us that he intended the encryption to use a deterministic encryption scheme [20] precisely to avoid using random values and the associated subliminal channel vulnerability. There is some risk in using this non-standard construction since the widely accepted minimum notion of security for public key encryption is IND-CPA, which requires a source of randomness.

itives use randomness to achieve semantic security [26], a strong notion of security for encryption schemes which guarantees that it is infeasible for adversaries to infer even partial information about the messages being encrypted (except maybe their length). Each choice for the random number allows a different valid ballot, which creates opportunities for subliminal channels.

Subliminal channels are easy to build in protocols or encryption schemes that use randomness. If a cryptographic protocol requests the DRE to choose a random number r and then publish it, the DRE can encode $|r|$ bits through judicious selection of r. Alternatively, given any randomized encryption scheme $e_k(\cdot, \cdot)$, the DRE can hide a bit b in an encryption of a message m by computing $c = e_k(m, r)$ repeatedly using a new random number r each time until the least significant bit of $h(c)$ is b. More generally, a malicious DRE can use this technique to hide ℓ bits in c with expected $O(2^\ell)$ work. Thus, all randomized encryption schemes contain subliminal channels.

Random subliminal channel attack. Neff's scheme uses randomness extensively. Each BMP consists of a pair of El Gamal ciphertexts, and the El Gamal encryptions are randomized. In forming the OVC, the DRE reveals half of the random values ω used in the encryptions (Figure 3.3).

For each BMP, one of the encryption pairs will be opened, revealing the random encryption parameter ω. This presents a subliminal channel opportunity.[3] Although the DRE must commit to the ballot before the voter chooses which side of the BMP to open, a malicious DRE can still embed $|\omega|$ bits of data for each BMP by using the same ω for both encryptions in the BMP. In this way ω is guaranteed to be revealed in the ballot.

This attack enables a high bandwidth subliminal channel in each voter's encrypted ballot.

[3] Another way a malicious DRE could embed a subliminal channel in Neff's scheme is if the voter doesn't choose all her unchoice challenges (i.e., the DRE is free to choose some of them). However, Neff outlines a variant of his proposal that solves this using two printers [60].

For example, in an election with 8 races and 5 candidates per race, there will be $40 \cdot \ell$ ballot mark pairs, where Neff suggests $\ell \geq 10$. A reasonable value of $|\omega|$ is 1024 bits. The total channel, then, can carry 128 bytes in each of the 400 BMPs, for a total of 51200 bytes of information per ballot. This is more than enough to leak the voter's choices and identifying information about the voter.

3.4.2 Mitigating random subliminal channels

Eschew randomness. One approach to prevent subliminal channels is to design protocols that don't require randomness. Designing secure protocols that do not use randomness is tricky, since so many proven cryptographic primitives rely on randomness for their security. Proposals relying on innovative uses of deterministic primitives, including Chaum's, deserve extra attention to ensure that forgoing randomness does not introduce any security vulnerabilities. Ideally, they would be accompanied by a proof of security.

Random tapes and their implementation. In a personal communication, Neff suggested that DREs could be provided with pre-generated tapes containing the random bits to use for all of their non-deterministic choices, instead of allowing them to choose their own randomness [59]. With a random tape for each BSN, the ballot becomes a deterministic function of the voter's choices and the random tape for that BSN. As long as the BSN is assigned externally before the voter selects her candidates, the ballots will be uniquely represented. This will eliminate the threat of random subliminal channels in encrypted ballots.

It is not enough for the intended computation to be deterministic; it must be verifiably so. Thus, we need a way to verify that the DRE has used the bits specified on the random tape, not some other bits. We present one possible approach to this problem using zero-knowledge (ZK) proofs [27] which allows everyone to verify that each DRE constructed ballots using the random numbers from its tape. We imagine that there are several optimizations to this approach which improve efficiency.

Suppose before the election, the trustees generate a series $r_{s,1}, r_{s,2}, \ldots$ of random values for each BSN s, and post commitments $C(r_{s,1}), C(r_{s,2}), \ldots$ on a public bulletin board. The election officials then load the random values $r_{s,1}, r_{s,2}, \ldots$ on the DRE which will use BSN s.

During the election, for each randomized function evaluation $f(r, \cdot)$, the DRE uses the next random value in the series and furnishes a ZK proof proving it used the next random value in the series. For example, in Neff's scheme, along with each $\boxed{b}$, which is an El Gamal encryption $e(r, b)$, the DRE includes a non-interactive zero knowledge proof of knowledge proving that 1) it knows a value $r_{s,i}$ which is a valid opening of the commitment $C(r_{s,i})$ and 2) $e(r_{s,i}, b) = \boxed{b}$. Verifying that each $r_{s,i}$ is used sequentially within a ballot enables any observer to verify that the encryption is deterministic, so there can be no random subliminal channels in $\boxed{b}$ or its opening $\textcircled{b}$.

However, there is a wrinkle to the above solution: under most schemes, constructing the zero-knowledge proof itself requires randomness, which creates its own opportunities of subliminal channels. It may be possible to determinize the ZK proof using research on unique zero-knowledge proofs (uniZK) [45, 46].

This approach may require further analysis to determine whether it is able to satisfy the necessary security properties.

Trusted hardware. Utilizing trusted hardware in DREs can also help eliminate subliminal channels. In this approach, the trusted hardware performs all computations that require random inputs and signs the encrypted ballot it generates. The signature enables everyone to verify the ballot was generated inside the trusted hardware. As long as trustees verify the DRE's trusted hardware is running the correct software and the trusted hardware isn't compromised, DREs will not be able to embed a random subliminal channel.

3.4.3 Multiple visual and semantic representations

A tabulator that accepts multiple equivalent visual or semantic representations of the voter's choice creates another subliminal channel opportunity. For example, if the tabulator accepts both James Polk and James␣␣Polk (with an extra space) as the same person, then a DRE can choose which version to print based on the subliminal channel bit it wants to embed.

Semantic subliminal channel attack. Chaum's scheme is vulnerable to multiple visual representations. A malicious DRE can create alternate ballot images for the same candidate that a voter will be unlikely to detect. Recall that Chaum's scheme encrypts an image of the ballot, and not an ASCII version of the voter's choices. The voter examines two transparencies together to ensure that the resulting image accurately represents their vote. A DRE could choose to use different fonts to embed subliminal channel information; the choice of font is the subliminal channel. To embed a higher bandwidth subliminal channel, the DRE could make minor modifications to the pixels of the ballot image that do not affect its legibility. Unless the voter is exceptionally fastidious, these minor deviations would escape scrutiny as the voter verifies the receipt. After mixing, the subliminal channel information would be present in the resulting plaintext ballots.

There is no computational cost for the DRE to embed a bit of information in the font. It can use a simple policy, such as toggling a pixel at the top of a character to encode a one, and a pixel at the bottom to encode a zero. On a 10 race ballot, using such a policy just once per word could embed 30 bits of information.

There is a qualitative difference between the semantic subliminal channels and the random subliminal channels. The information in the semantic channels will only become apparent after the mix net decrypts the ballot since the channel is embedded in the plaintext of the ballot. In contrast, the random subliminal channels leak information when the ballots are made available on the bulletin board.

Mitigation. To prevent the semantic subliminal channel attack, election officials must establish official unambiguous formats for ballots, and must check all ballots for conformance to this approved format. Any deviation indicates a ballot produced by a malicious DRE. Such non-conforming ballots should not be allowed to appear on the bulletin board, since posting even a single suspicious ballot on the bulletin board could compromise the privacy of all voters who used that DRE. Unfortunately, the redaction of such deviant ballots means that such ballots in will not be able to be verified by the voter through normal channels.

An even more serious problem is that this policy violates assumptions made by the mix net. One would need to ensure the mix net security properties still hold when a subset of the plaintexts are never released.

The order in which ballots appear will also need to be standardized. Otherwise, a DRE can choose a specific ordering of ballots on the public bulletin board as a low bandwidth subliminal channel [42]. Fortunately, it is easy to sort or otherwise canonicalize the order of ballots before posting them publicly.

3.4.4 Discussion

Subliminal channels pose troubling privacy and voter coercion risks. In the presence of such attacks, we are barely better off than if we had simply posted the plaintext ballots on the bulletin board in unencrypted form for all to see. The primary difference is that subliminal channel data may be readable only by the malicious parties. This situation seems problematic, and we urge protocol designers to design voting schemes that are provably and verifiably free of subliminal channels.

3.5 Denial of service attacks and election recovery

Although Neff's and Chaum's schemes can detect many attacks, recovering legitimate election results in the face of these attacks may be difficult. In this section, we present several detectable but irrecoverable denial of service (DoS) attacks launched at different stages of the voting and tallying process. We consider attacks launched by malicious DREs and attacks launched by malicious tallying software, and discuss different recovery mechanisms to resist these attacks.

3.5.1 Denial of service (DoS) attacks

Launched by malicious DREs. Malicious DREs can launch several DoS attacks which create detectable, but unrecoverable situations. We present two classes of attacks: ballot deletion and ballot stuffing.

In a ballot deletion attack, a malicious DRE erases voters' ballots or submits random bits in their place. Election officials and voters can detect this attack after the close of polls, but there is little they can do at that point. Since the electronic copy serves as the only record of the election, it is impossible to recover the legitimate ballots voted on that DRE.

DREs can launch more subtle DoS attacks using ballot stuffing. Recall that both Neff's and Chaum's schemes use ballot sequence numbers (BSNs) to uniquely identify ballots. BSNs enable voters to find and verify their ballots on the public bulletin board, and by keeping track of the set of valid BSNs, election officials can track and audit ballots.

In the *BSN duplication attack,* a DRE submits multiple ballots with the same BSN. Election officials will be able to detect this attack after the ballots reach the bulletin board, but recovery is difficult. It is not clear how to count ballots with the same BSN. Suppose a DRE submits 100 valid ballots (i.e., from actual voters) and 100 additional ballots, using the same BSN for all the ballots. How do talliers distinguish the invalid ballots from the valid ones?

In the *BSN stealing attack,* a malicious DRE "steals" BSNs from the set of BSNs it would normally assign to legitimate voters' ballots. For a particular voter, the DRE might submit a vote of its own choosing for the BSNit is supposed to use, and on the voter's receipt print a different (invalid) BSN. Since the voter will not find her ballot on the bulletin board, this attack can be detected, but recovery is tricky: how do election officials identify the injected ballots and remove them from the tally?

Neff's and Chaum's scheme enable voters and/or election officials to detect these attacks, but recovery is non-trivial because 1) the voters' legitimate ballots are missing and 2) it is hard to identify the invalid ballots injected by the DRE.

Launched by malicious tallying software. DoS attacks in the tallying phase can completely ruin an election. For example, malicious tallying softwares can delete the trustees' keys, making decryption and tallying of the encrypted ballots forever impossible. Malicious bulletin board software can erase, insert, or delete ballots.

Selective DoS. An attacker could use DoS attacks to bias the outcome of the election. Rather than ruining the election no matter its outcome, a more subtle adversary might decide whether to mount a DoS attack or not based on who seems to be willing the race. If the adversary's preferred candidate is winning, the adversary need do nothing. Otherwise, the adversary might try to disrupt or ruin the election, forcing a re-election and giving her preferred candidate a second chance to win the election, or at least raising questions about the winner's mandate and reducing voters' confidence in the process.

There are many ways that selective DoS attacks might be mounted:

- If an outsider has a control channel to malicious DREs, the outsider could look at the polls and communicate a DoS command to the DREs.
- An autonomous DRE could look at the pattern of votes cast during the day, and fail (deleting all votes cast so far at that DRE) if that pattern leans towards the undesired candidate. This would disrupt votes cast only in precincts leaning against the attacker's preferred candidate.
- If trustees' software is malicious, it could collude to see how the election will turn out, then cause DoS if the result is undesirable. Note that if all trustees are running the same tallying software, this attack would require only a single corrupted programmer.

Selective DoS attacks are perhaps the most troubling kind of DoS attack, because they threaten election integrity and because attackers may have a real motive to launch them.

3.5.2 Mitigation strategies and election recovery

Note that in all these attacks, non-malicious hardware or software failures could cause the same problems. This may make it hard to distinguish purposeful attacks from unintentional failures.

The above attacks create irrecoverable situations because voters' legitimate ballots are lost or corrupted, the bulletin board contains unidentifiable illegitimate ballots submitted by malicious DREs, or both. In this section, we evaluate two recovery mechanisms for these DoS attacks: *revoting* and a *voter verified paper audit trail*.

Revoting. One recovery strategy is to allow cheated voters to revote. Depending on the scope of the attack or failure, this could range from allowing only particular voters to revote to completely scrapping the election and starting over. However, revoting is problematic. Redoing the entire election is the most costly countermeasure. Alternatively, election officials could allow only those voters who have detected cheating to revote. Unfortunately, this is insufficient. Less observant voters who were cheated may not come forward, and it may be hard to identify and remove illegitimate ballots added by a malicious DRE. Revoting does not help with selective DoS.

Voter verified paper audit trail. A voter verified paper audit trail (VVPAT) system produces a paper record verified by the voter before her electronic ballot is cast [51]. This paper record is cast into a ballot box. The paper trail is an official record of the voter's vote but is primarily intended for use in recounts and auditing.

It would not be hard to equip cryptographic voting systems with a VVPAT. This would provide a viable mechanism for recovering from DoS attacks. In addition to providing an independent record of all votes cast, VVPAT enables recovery at different granularities. If election officials conclude the entire electronic record is questionable, then the entire VVPAT can be counted. Alternatively, if only a single precinct's electronic record is suspect, then this precinct's VVPAT record can be counted in conjunction with the other precincts' electronic records. This approach enables officials to keep the universal verifiability of the uncorrupted precincts while recovering the legitimate record of the corrupted precinct.

A third benefit of VVPAT is that it provides an independent way to audit that the cryptography is correctly functioning. This would be one way to help all voters, even those who do not understand the mathematics of these cryptographic schemes, to be confident that their vote will be counted correctly.

3.6 Implementing secure cryptographic voting protocols

A secure implementation of Neff and Chaum's protocol will still need to resolve many issues. In this section, we outline important areas that Neff and Chaum have not yet specified. These parts of the system need to be fully designed, implemented, and specified before one can perform a comprehensive security review. Also, we list three open research problems which we feel are important to the viability of these schemes.

3.6.1 Underspecifications

Bulletin board. Both protocols rely on a public bulletin board to provide anonymous, read only access to the data. The data must be stored robustly, overcoming software and mechanical failures as well as malicious attacks. Further, only authenticated parties should be able to append messages to the bulletin board. An additional requirement is to ensure that the system delivers the same copy of the bulletin board contents to each reader. If the bulletin board were able to discern a voter's identity, say by IP address, it could make sure the voter always saw a mix transcript that included a proof that their vote was counted. But, for the official transcript, the mix net and bulletin board could collude to omit the voter's ballot. In this scenario, the voter would think her vote had been counted but in reality it was not.

Neff and Chaum have not yet elaborated on a proposed bulletin board architecture or the properties they require. We imagine that the principles of distributed storage systems, such as Farsite, CFS, or OceanStore [3, 22, 70], might be applicable in the bulletin board setting. However, without a further specification of exactly which architecture would be used, we cannot evaluate the system's security.

BSN assignment. Neff's and Chaum's schemes do not specify how to assign BSNs to voters' ballots. BSNs could be assigned externally by a smartcard initializer which authorizes a voter to use a DRE, or be assigned by DREs, say by a monotonically increasing counter prefixed by the DRE machine ID.[4] Clever BSN assignment combined with careful auditing and sign-in procedures could help limit the scope of some of the DoS attacks in Section 3.5, but since DREs can always erase or corrupt a voter's electronic ballot after she casts it, we still must consider recovery mechanisms.

Tallying software. Both Neff's and Chaum's schemes treat the tallying software as a black box. We surmise, that it, too, has stringent requirements on its correct implementation. If all trustees use tallying software from a single source, then this software might collude without the trustees' knowledge and invalidate the system's integrity guarantees. Though n-version programming might be able to counter this threat, it makes software development very expensive and requires detailed interface specifications to ensure that all versions of the software will interoperate. We have not seen any details on how to ensure that the tallying software cannot collude.

[4]David Chaum later conveyed to us that he intended his scheme to use a counter to assign BSNs [20].

3.6.2 Open research problems

Subliminal channels. Developing cryptographic protocols that address subliminal channels would help resist privacy and coercion attacks. Subliminal channels in the ballots subvert the confidentiality guarantees provided by encryption. We present some techniques in Section 3.4 to eliminate subliminal channels in encrypted ballots, but we believe this is still an area for future research.

Mix net security models. We would like to see a definition of security for mix nets that is comprehensive for the voting setting. Such a definition must be natural enough to inspire confidence that it is the correct model. For instance, Jakobsson illustrates a subtle privacy violation if the encryption used in the mixes do not provide non-malleability [32], and others have shown similar results [66]. This illustrates the importance and non-triviality of formulating a correct security model for mix nets. We believe the security of cryptographic voting systems would benefit from a thorough study of the relationship between the mix net requirements and those of the rest of the system.

Humans as protocol participants. These voting protocols require voters to not just use a cryptographic system, but also to participate in a cryptographic protocol. Cryptographic protocols are fragile to deviations and mistakes in their implementation, and humans have been known to make mistakes. A high level understanding of the protocol is not sufficient; to minimize errors, voters often need to understand how the protocol works. Alternatively, voting protocols must be designed to be as resilient as possible to mistakes made by the average voter. Voter education could help, but this raises an important human-computer interaction problem: how do we educate voters about these issues without discouraging them that these systems are too complicated to securely use?

3.7 Conclusion

We laud Neff's and Chaum's ambitious goal: developing a coercion free, privacy preserving voter-verifiable election system. Their systems represent a significant security improvement over current DRE-based paperless systems. Neff's and Chaum's schemes also strive to limit reliance on trusted software and hardware. Most notably, these schemes do not require voters to trust DREs since voters can detect malicious behavior.

Neff's and Chaum's schemes are fully specified at the cryptographic protocol level, but they are underspecified from the systems and human interaction level. Due in part to this underspecification, we have discovered a number of potential weaknesses which only became apparent when considered in the context of an entire voting system. We expect that a well designed implementation and deployment may be able to mitigate or even eliminate the impact of these weaknesses.

We found solutions for some of these weaknesses, but we also identified new challenges and open problems for electronic voting systems. First, subliminal channels have the potential to erode voter privacy and enable voter coercion. Any system that uses a public bulletin board must ensure that the ballots it posts have a unique representation. Second, these voting protocols present a new research challenge by placing human voters directly within an interactive cryptographic protocol. Protocol designers have previously assumed participants are infallible computer agents, but voting protocols must cope with human error and ignorance.

Despite these challenges, we are optimistic about the future prospects of these voting systems.

Chapter 4

Privacy

In this chapter, we study the privacy problem, identify several challenges in assuming the secrecy of the ballot, and then propose techniques to address these issues. We start with a generalized description of a voting session, and then describe how privacy can be violated in a range of different voting technologies. Finally, we describe a general approach to prevent privacy violations. We outline our novel solution in this chapter, and in Chapter 5, apply this generalized solution to one particular voting implementation in further detail.

4.1 Voting sessions

A typical voting machine is used many times throughout the day by many voters. Cost, space, and manageability constraints necessitate reusing voting machines throughout the day since we cannot afford to give each voter their own voting machine. This creates the real possibility of voter information flowing from one voter to another through the voting machine.

We define a *voting session* as one voter's interactions with the voting machine. Recalling

Section 2.1, all voting sessions are encompassed within the active voting phase. A voting session starts with the voter's first use of a particular voting machine and ends when they leave the voting machine. It is assumed that only one voter uses the machine during each session. After each voting session, the machine returns to a start state and readies itself for the next voter's session.

4.2 Avenues for information flows

In this section, we look at different voting technologies and highlight some of the ways privacy violations might occur. Table 4.1 summarizes the ways that private information might leak out of the machine as well as the relative severity of the potential leak.

4.2.1 DRE

A voting session with a DRE begins with the voter presenting their authentication token and ends after they make their selections, confirm the choices, and leave the voting machine. A DRE has many output devices: the voting screen, audio output, and the electronic ballot box. DREs with VVPAT [51] contain also have a printer for the paper receipt. Each of these output devices presents a different avenue for data to leak.

With corrupt software, a DRE could reveal previous voters' selections to the screen. Just as in Section 3.4, the malicious DRE could reveal the ballot casting times for all ballots for a specific candidate. Correlating this information with when voters leave the polling booth easily reveals voters' choices. A party could activate malicious code to gain access to this confidential data with a specific and unusual sequence of inputs. Assume that each vote can be represented with a four or five bits, or alternatively one ASCII character; with a ballot of 100 races, a single voter's

Voting Technology	Output Channel	Flow capacity	Notes
DRE	Screen	Large	
	VVPAT printed record	Medium	
	Audio accessibility interface	Small	
	Vote storage	Large	We can prevent leaks using [55]
Cryptographic voting protocols	Receipt	Medium	
	Screen	Large	
	Audio accessibility interface	Small	
	Bulletin board	Large	Can be read anonymously over the Internet
	Vote storage	Large	We can prevent leaks using [55]
Ballot marking device	Screen	Large	
	Marked ballot	Large	
Optical scan reader	Confirmation screen	Small	
	Vote storage	Large	We can prevent leaks using [55]

Table 4.1: Ways that prior vote information might escape from a voting machine in different voting technologies.

choices can fit in one line of text. This means that over 100 voters' full ballots can fit onto two pages of text. It would be inconceivable to copy two full pages of ASCII gibberish down by hand, but a digital camera would be a convenient tool to download the data from the DRE.

The audio output device, used to improve accessibility for voters with visual impairments, can also be used to surreptitiously leak prior voters' data. A malicious DRE could simply read out prior voter's selections. However, this is a slow process, so it is infeasible to quickly leak all prior voters' data.

DREs store their ballots into an electronic ballot box. This is usually a removable memory device that is used for summing the votes cast on the DRE. Depending upon the voting jurisdiction's procedures, the contents of the ballot box may be made public. This represents a large potential vehicle for information leakage. The ballot box 1) may contain extraneous data that reveals voters' selections in unused portions of the ballot box device; or 2) may encode hidden data using the order the elements are on disk. These allow a malicious voting machine to leak casting time of all of the votes. Using a standardized data format and the techniques developed in conjunction with Molnar et al [55], it is possible to eliminate privacy leaks from a electronic ballot boxes.

Finally, some DREs are being equipped with VVPAT printers. Even though the voter does not keep or even touch the paper record, it represents an output channel to convey private information. The paper record displays the entire list of a voter's selections. After reviewing the printed voter record, the machine queries the voter and either prints an acceptance note on the record, or a spoil note and allows the voter to edit their response and again review the printed ballot. Since the printed record is retained by election officials and could undergo later scrutiny, a malicious DRE must attempt to disguise private data it is conveying. One way for the DRE to leak a prior voter's choices is by printing the selections to the printer. So as not to arouse suspicion with extra votes in the paper record, the DRE could then spoil the printed record, mimicking a voter who changed their mind.

4.2.2 Cryptographic voting protocol

Chapter 3 addresses cryptographic voting protocols and the privacy risks that they face. The ones discussed there augment normal DREs, and so inherit their risks, as well as new ones through the bulletin boards. The receipt the voter takes with them could also contain private data detailing prior voters' selections. The amount of data is bounded by the receipt's size, but a clever encoding can leak a substantial amount of data. The systems described in that chapter are based on DREs, so have similar privacy problems with the screen and vote storage mechanisms.

4.2.3 Ballot marking device

Ballot marking devices are similar to DREs in presenting their choices, but instead of storing an electronic ballot, they print a ballot readable both by machines and humans. The voter reviews the printed ballot and upon accepting it takes it to the optical scan reader that scans the ballot and stores a paper copy, or discards the printed ballot and enters new selections.

Since voters interact with ballot marking devices in a similar fashion to DREs, it is natural that ballot marking devices face similar privacy vulnerabilities. Just as a malicious DRE can use the screen or audio interface to convey private information, so can ballot marking devices.

The printer portion of the ballot marking device can be used for an even simpler attack. The printer can be used to print a summary of all prior voter's decisions. A suitable encoding, such as the one from Section 4.2.1, can easily fit all voter data on a page or two. To avoid suspicion, the ballot marking device would also print a legitimate ballot for casting.

The data obtained from a ballot marking device is less reliable than the data from either a DRE or a cryptographic protocol solutions. Since the ballot marking device is used for ballot preparation, but not casting, it can only possibly know which ballots have been prepared for casting. Its knowledge of prepared ballots is a superset of the actual cast ballots. These attacks should prove damaging to privacy, but it may not always be possible to know with absolute certainty how someone voted.

4.2.4 Optical scan reader

Optical scan machines accept ballots created either with a ballot marking device or by a person on a paper. Voters feed their ballots to the machine, which scans it and checks for errors (blank ballots and overvotes, for example). Finding an error, it displays a short message on its LCD screen and rejects the ballot; or if the ballot is error free, it scans the paper ballot and stores the ballot in a locked ballot box.

Like DREs, optical scan machines contain a removable memory device that is counted at the precinct. This represents a potentially large privacy channel. But as in the DRE case, fixed ballot formats and techniques from Molnar et al. can prevent privacy leaks [55].

The small LCD screen presents another output channel. However, the likelihood of it being exploited is small. Election officials must provide access control to the optical scan reader to prevent voters from adding multiple ballots at once. Under the election officials watch, an adversary spending lots of time memorizing the LCD screen, taking pictures, or writing copious notes, would attract attention. For this reason, the optical scanner's LCD is not a major threat.

4.3 Reboots

One of the assumptions we started with, and a key enabler for privacy leaks, is the need to multiplex voting hardware among multiple voters. Suppose, instead, that each voter uses a fresh voting machine at the polling station. If the machines are not networked, it becomes trivial to see that a machine cannot leak data about prior voters, since the machine starts without any vote state, and is only used by a single voter. This approach, though, is not feasible since it dramatically increases the cost for hardware as well as the cost for managing the infrastructure.

Our approach seeks to blend the purity of individual voter machines with the cost and maintenance benefits of shared DREs.

A traditional DRE, for example the Diebold AccuVote-TS, is designed to run as a single operating system process. The functions of the DRE—validating the voter, presenting choices, confirming those choices, storing the ballot, and administrative functions—are all a part of the same address space.

Let us examine one particular strategy we can use to better verify Property 1, which requires that one voter's selections must not influence the voting experience observed by the next voter. Suppose after every voter has voted, the voting machine is turned off and then restarted. This is enough to ensure that the voting machine's memory will not contain any information about the prior voter's selections when it starts up. Of course, the prior voter's selections must still be recorded on permanent storage (e.g., on disk) for later counting, so we also need some mechanism to prevent the machine from reading the contents of that storage medium. One conservative strategy would be to simply require that any file the voting machine writes to must always be opened in write-only mode, and should never be opened for reading. More generally, we can allow the voting machine to read from some files, such as configuration files, as long as it does not have the ability to write to them. Thus the set of files on permanent storage are partitioned into two classes: a set of read-only files (which cannot be modified by the voting machine), and a set of write-only files (which cannot be read by the voting machine). To summarize, our strategy for enforcing Property 1 involves two prongs:

1. Ensure that a reboot is always triggered after a voter ends their session.

2. Check every place a file can be opened to ensure that data files are write-only, and configuration files are read-only.

There must still be a mechanism to prevent the voting machine from overwriting existing data, even if it cannot read that data.

Rebooting also helps to ensure that all voters are treated equally. After rebooting, the memory is reset to a known state. This observation was made by Candea et al., who relied on the idea to eliminate bugs that creep up unpredictably [16, 17].

We emphasize this design strategy is not the only way to verify this particular property. Rather, it is one technique we can implement that reduces the problem of enforcing Property 1 to the problem of enforcing a checklist of easier-to-verify conditions that suffice to ensure Property 1 will always hold.

4.3.1 Applicability

While Candea et al. [16, 17] use prophylactic reboots to increase availability, we believe the strategy of rebooting for privacy is novel. It is also broadly applicable to all of the different technologies mentioned, including DREs, DREs with cryptographic voting protocols, ballot marking devices, as well as optical scan readers. In each case, the reboot erases the memory between each voter session, guaranteeing the voter starts with a known and clean memory state. After ensuring the only session-writable storage is read only, it is simple to verify prior voters' private information cannot be leaked at the voting machine.

In Chapter 5, we demonstrate the viability of the reboot approach to a specific platform to guarantee Property 1 in addition to other properties.

Chapter 5

Designing voting machines for verification

In this chapter, we provide techniques to help vendors, independent testing agencies, and others verify critical security properties in direct recording electronic (DRE) voting machines. We expand upon the privacy preserving techniques presented in Chapter 4 to address Property 1 and also address Property 2 to guarantee a ballot is only cast with the voter's consent. With a little additional work, the other properties are amenable to our techniques. We rely on specific hardware functionality, isolation, and architectural decisions to allow one to easily verify critical security properties. We believe our techniques will help us verify other properties as well though we have not demonstrated this. Verification of these security properties is one step towards a fully verified voting machine.

Parts of this work are drawn with permission from previously published work [74].

5.1 Introduction

In this chapter we seek to answer how can we reason about, or even prove, relevant security properties in voting machines. As we have seen, the flurry of reports criticizing the trustworthiness of direct recording electronic (DRE) voting machines, computer scientists have not been able to allay voters' concerns about this critical infrastructure [42, 18, 72, 90]. The problems are manifold: poor use of cryptography, buffer overflows, and in at least one study, poorly commented code.

The ultimate security goal would be a system where any voter, without any special training, could easily convince themselves about the correctness of *all* relevant security properties. Our goal is not so ambitious; we address convincing those with the ability to understand code the correctness of a few security properties. For clarity, we focus on two important security properties in this chapter. These properties were originally described in Chapter 2. Briefly, recall that Property 1 states that a voter's interactions should not affect any subsequent voter's sessions. Property 2 states that a ballot should not be cast without the voter's consent. Verification of these properties, as well as the others we described in Chapter 2, are a step towards the full verification of a voting machine.

Current DREs are not amenable to verification of these security properties; for instance, version 4.3.1 of the Diebold AccuVote-TS electronic voting machine consists of 34 712[1] lines of vendor-written C++ source code, all of which must be analyzed to ensure Properties 1 and 2. One problem with current DRE systems, in other words, is that the trusted computing base (TCB) is simply too large. The larger problem, however, is the code simply is not *structured* to verify security

[1]Kohno et al. count the total number of lines in their paper [42]; for a fair comparison with our work, we look at source lines of code, which excludes comments and whitespace from the final number. Hence, the numbers cited in their paper differ from the figure we list.

properties.

In this chapter, we develop a new architecture that significantly reduces the size of the TCB for verification of these properties. Our goal is to make voting systems more amenable to efficient verification, meaning that implementations can be verified to be free of malicious logic. By appropriate architecture design, we reduce the amount of code that would need to be verified (e.g., using formal methods) or otherwise audited (e.g., in an informal line-by-line source code review) before we can trust the software, thereby enhancing our ability to gain confidence in the software. We stress that our architecture assumes voters will be diligent: we assume that each voter will closely monitor their interaction with the voting machines and look for anomalous behavior, checking (for example) that her chosen candidate appears in the confirmation page.

We present techniques that we believe are applicable to DREs. We develop a partial voting system, but we emphasize that this work is not complete. As we discussed in Section 2.1, voting systems comprise many different steps and procedures: pre-voting, ballot preparation, audit trail management, post-election, recounts, and an associated set of safeguard procedures. Our system only addresses the active voting phase. As such, we do *not* claim that our system is a replacement for an existing DRE or a DRE system with a paper audit trail system. See Section 5.6 for a discussion of using paper trails with our architecture.

Technical elements of our approach. We highlight two of the key ideas behind our approach. First, we focus on creating a trustworthy vote confirmation process. Most machines today divide the voting process into two phases: an initial vote selection process, where the voter indicates who they wish to vote for; and a vote confirmation process, where the voter is shown a summary screen listing their selections and given an opportunity to review and confirm these selections before casting their ballot. The vote selection code is potentially the most complex part of the system, due to the need for complex user interface logic. However, if the confirmation process is easy to verify, we can verify many important security properties without analyzing the vote selection process. Our architecture splits the vote confirmation code into a separate module whose integrity is protected using hardware isolation techniques. This simple idea greatly reduces the size of the TCB and means that only the vote confirmation logic (but not the vote selection logic) needs to be examined during a code review for many security properties, such as Property 2.

Second, we use hardware resets to help ensure Property 1, as originally outlined in Section 4.3. In our architecture, most modules are designed to be stateless; when two voters vote in succession, their execution should be independent. We use hard resets to restore the state of these components to a consistent initial value between voters, eliminating the risk of privacy breaches and ensuring that all voters are treated equally by the machine.

Our architecture provides several benefits. It preserves the voting experience that voters are used to with current DREs. It is compatible with accessibility features, such as audio interfaces for voters with visual impairments, though we stress that we do not implement such features in our prototype. It can be easily combined with a voter-verified paper audit trail (VVPAT). Our prototype implementation contains only 5 085 lines of trusted code.

5.2 Goals and assumptions

Security goals. For clarity, in this chapter we focus on enabling efficient verification of Properties 1 and 2 (see Chapter 2), though we hope to enable the efficient verification of other properties as well. Property 1 reflects a privacy goal: an adversary should not be able to learn any information about how a voter voted besides what is revealed by the published election totals. Property 2 reflects an integrity goal: even in the presence of an adversary, the DRE should record the voter's vote exactly as the voter wishes. Further, an adversary should not be able to undetectably alter the vote once it is stored. We wish to preserve these properties against the classes of adversaries discussed below.

Wholesale and retail attacks. A wholesale attack is one that, when mounted, has the potential of affecting a broad number of deployed DREs. A classic example might be a software engineer at a major DRE manufacturer inserting malicious logic into her company's DRE software. Prior work has provided evidence that this it is a concern for real elections [5]. Such an attack could have nationwide impact and could compromise the integrity of entire elections, if not detected. Protecting against such wholesale attacks is one of our primary goals. In contrast, a retail attack is one restricted to a small number of DREs or a particular polling location. A classic retail attack might be a poll worker stuffing ballots in a paper election, or selectively spoiling ballots for specific candidates.

Classes of adversaries. We desire a voting system that:

- Protects against *wholesale* attacks by election officials, vendors, and other insiders.
- Protects against *retail* attacks by insiders when the attacks *do not* involve compromising the physical security of the DRE or the polling place (e.g., by modifying the hardware or software in the DRE or tampering with its surrounding environment).
- Protects against attacks by outsiders, e.g., voters, when the attacks *do not* involve compromising physical security.

We explicitly do not consider the following possible goals:

- Protect against *retail* attacks by election insiders and vendors when the attacks *do* involve compromising physical security.

- Protect against attacks by outsiders, e.g., voters, when the attacks *do* involve compromising physical security.

On the adversaries that we explicitly do not consider. We explicitly exclude the last two adversaries above because we believe that adversaries who can violate the physical security of the DRE will always be able to subvert the operation of that DRE, no matter how it is designed or implemented. Also, we are less concerned about physical attacks by outsiders because they are typically *retail attacks*: they require modifying each individual voting machine one-by-one, which is not practical to do on a large scale. For example, to attack privacy, a poll worker could mount a camera in the voting booth or, more challenging but still conceivable, an outsider could use Tempest technologies to infer a voter's vote from electromagnetic emissions [43, 88]. To attack the integrity of the voting process, a poll worker with enough resources could replace an entire DRE with a DRE of her own. Since this attack is possible, we also do not try to protect against a poll worker that might selectively replace internal components in a DRE. We assume election officials have deployed adequate physical security to defend against these attacks.

We assume that operating procedures are adequate to prevent unauthorized modifications to the voting machine's hardware or software. Consequently, the problem we consider is how to ensure that the original design and implementation are secure. While patches and upgrades to the voting system firmware and software may occasionally be necessary, we do not consider how to securely distribute software, firmware, and patches, nor do we consider version control between components.

Attentive voters. We assume that voters are attentive. We require voters to check that the votes shown on the confirmation screen do indeed accurately reflect their intentions; otherwise, we will not be able to make any guarantees about whether the voter's ballot is cast as intended. Despite our reliance on this assumption, we realize it may not hold for all people. Voters are fallible and not all will properly verify their choices. To put it another way, our system offers voters the *opportunity* to verify their vote. If voters do not take advantage of this opportunity, we cannot help them. We do not assume that all voters will avail themselves of this opportunity, but we try to ensure that those who do, are protected.

5.3 Architecture

We focus this chapter on our design and implementation of the "active voting" phase of the election process (cf. Figure 2.1). We choose to focus on this step because we believe it to be one of the most crucial and challenging part of the election, requiring interaction with voters and the ability to ensure the integrity and privacy of their votes. We remark that we attempt to reduce the trust in the canvassing phase by designing a DRE whose output record is both privacy-preserving (anonymized) and integrity-protected.

5.3.1 Architecture motivations

To see how specific design changes to traditional voting architectures can help verify properties, we will go through a series of design exercises starting from current DRE architectures

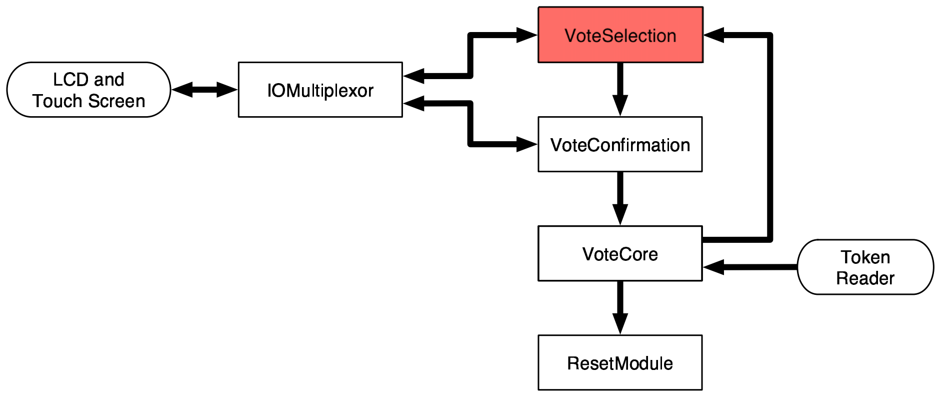


Figure 5.1: Our architecture, at an abstract level. For the properties we consider, the VoteSelection module need not be trusted, so it is colored red.

and finishing at our design. The exercises will be motivated by trying to design a system that clearly exhibits Properties 1 and 2.

Resetting for independence. Chapter 4 highlights our approach to achieving privacy in a DRE. Recall, to satisfy the conditions of the approach, two conditions must be met:

1. Ensure that a reboot is always triggered after a voter ends their session.
2. Check every place a file can be opened to ensure that data files are write-only, and configuration files are read-only.

For our architecture, we introduce a separate component whose sole job is to manage the reset process. The BallotBox triggers the ResetModule after a ballot is stored. The reset module then reboots a large portion of the DRE and manages the startup process. We use a separate component so that it is simple to audit the correctness of the ResetModule.

Isolation of confirmation process. In considering Property 2, which requires the voter's consent to cast in order for the ballot to be stored, we will again see how modifying the DRE's architecture in specific ways can help verify correctness of this property.

The consent property in consideration requires auditors to confidently reason about the casting procedures. An auditor (perhaps using program analysis tools) may have an easier time reasoning about the casting process if it is isolated from the rest of the voting process. In our architecture, we take this approach in combining the casting and confirmation process, while isolating it from the vote selection functionality of the DRE. With a careful design, we only need to consider this sub-portion to verify Property 2.

From our DRE design in the previous section, we introduce a new component, called the VoteConfirmation module. With this change, the voter first interacts with a VoteSelection module that presents the ballot choices. After making their selections, control flow passes to the VoteConfirmation module that performs a limited role: presenting the voter's prior selections and then waiting for the voter to either 1) choose to modify their selections, or 2) choose to cast their ballot. Since the VoteConfirmation module has limited functionality, it only needs limited support for GUI code; as we show in Section 5.5.1 we can more easily analyze its correctness since its scope is limited. If the voter decides to modify the ballot, control returns to the VoteSelection module.

Note the voter interacts with two separate components: first the VoteSelection component and then VoteConfirmation. There are two ways to mediate the voter's interactions with the two components: 1) endow each component with its own I/O system and screen; 2) use one I/O system and a trusted I/O "multiplexor" to manage which component can access the screen at a time. The latter approach has a number of favorable features. Perhaps the most important is that it preserves the voter's experience as provided by existing DRE systems. A voting machine with two screens requires voters to change their voting patterns, and can introduce the opportunity for confusion or even security vulnerabilities. Another advantage is cost: a second screen adds cost and complexity. One downside is that we must now verify properties about the IOMultiplexor. For example, it must route the input and output to the proper module at the appropriate times.

In the the final piece of our architecture, we introduce a VoteCore component. After the voter interacts with the VoteSelection system and then the VoteConfirmation module to approve their selection, the VoteCore component stores the ballot on indelible storage in its BallotBox and then cancels the voter's authentication token. Then, as we described above, it initiates a reset with the ResetModule to clear the state of all modules.

Let us return to our original property: how can we verify that a ballot can only be cast with the voter's approval? With our architecture, it suffices to verify that:

1. A ballot can only enter the VoteCore through the VoteConfirmation module.

2. The VoteCore gives the voter the opportunity to review the exact contents of the ballot.

3. A ballot can only be cast if the voter unambiguously signals their intent to cast.

To prove the last condition, we add hardware to simplify an auditor's understanding of the system, as well as to avoid calibration issues with the touch screen interface. A physical cast button, enabled only by the confirmation module, acts as a gate to stop the ballot between the VoteSelection and VoteCore modules. The software in the VoteConfirmation module does not send the ballot to the VoteCore until the CastButton is depressed; and, since it is enabled only in the VoteConfirmation module, it is easy to gain assurance that the ballot cannot be cast without the voter's consent. Section 5.5.1 will show how we achieve this property based on the code and architecture.

There is a danger if we must adjust the system's architecture to meet each particular security property: a design meeting all security properties may be too complex. However, we designed the architecture with the other security properties from Chapter 2 in mind. Isolating the confirmation process is a key insight that can simplify verifying other properties. The confirmation process is at the heart of many properties, and a small, easily understood confirmation process helps not just in verifying Property 2. For other properties, we rely on software verification, as described in Chapter 6.

5.3.2 Detailed module descriptions

Voter authentication. After a voter signs in at a polling station, an election official would give that voter a voting token. In our implementation, we use a magnetic stripe card, but the token could also be a smartcard or a piece of paper with a printed security code. Each voting token is valid for only one voting machine. To begin voting, the voter inserts the token into the designated voting machine. The VoteCore module reads the contents of the token and verifies that the token is designated to work on this machine (via a serial number check), is intended for this particular election, has not been used with this machine before, and is signed using some public-key signature scheme. If the verification is successful, the VoteCore module communicates the contents of the voting token to the VoteSelection module.

Vote selection. The VoteSelection module parses the ballot definition file and interacts with the voter, allowing the voter to select candidates and vote on referenda. The voting token indicates which ballot to use, e.g., a Spanish ballot if the voter's native language is Spanish or a Democratic ballot if the voter is a Democrat voting in a primary. The VoteSelection module is intended to follow the rules outlined in the ballot definition file, e.g., allowing the voter to choose up to three candidates or to rank the candidates in order of preference. Of course, the VoteSelection module is untrusted and may contain malicious logic, so there is no guarantee that it operates as intended. The VoteSelection module interacts with the voter via the IOMultiplexor.

Vote confirmation. After the voter is comfortable with her votes, the VoteSelection module sends a description of the voter's preferences to the VoteConfirmation module. The VoteConfirmation module interacts with the voter via the IOMultiplexor, displaying a summary screen indicating the current selections and prompting the voter to approve or reject this ballot. If the voter approves, the VoteConfirmation module sends the ballot image[2] to the VoteCore module so it can be recorded. The VoteConfirmation module is constructed so that the data that the VoteConfirmation module sends to the VoteCore module is exactly the data that it received from the VoteSelection module.

Storing votes and canceling voter authentication tokens. After receiving a description of the votes from the VoteConfirmation module, the VoteCore atomically stores the votes and cancels the voter authentication token. Votes are stored on a durable, history-independent, tamper-evident, and subliminal-free vote storage mechanism [55]. By "atomically," we mean that once the VoteCore component begins storing the votes and canceling the authentication token, it will not be reset until after those actions complete. After those actions both complete, the VoteCore will trigger a reset by sending a message to the ResetModule. Looking ahead, the only other occasion for the ResetModule to trigger a reset is when requested by VoteCore in response to a user wishing to cancel her voting session.

[2]A *ballot image* is merely a list of who this voter has voted for. It need not be an actual image or picture.

Cleaning up between sessions. Upon receiving a signal from the VoteCore, the ResetModule will reset all the other components. After those components awake from the reset, they will inform the ResetModule. After all components are awake, the ResetModule tells all the components to start, thereby initiating the next voting session and allowing the next voter to vote. We also allow the VoteCore module to trigger a reset via the ResetModule if the voter decides to cancel their voting process; when a voter triggers a reset in this way, the voter's authentication token is not canceled and the voter can use that token to vote again on that machine at a later time. Although the VoteCore has access to external media to store votes and canceled authentication tokens, all other state in this component is reset.

Enforcing a trusted path between the voter and the VoteConfirmation module. Although the above discussion only mentions the IOMultiplexor in passing, the IOMultiplexor plays a central role in the security of our design. Directly connecting the LCD and touch screen to both the VoteSelection module and the VoteConfirmation module would be unsafe: it would allow a malicious VoteSelection module to retain control of the LCD and touch screen forever and display a spoofed confirmation screen, fooling the voter into thinking she is interacting with the trusted VoteConfirmation module when she is actually interacting with malicious code. The IOMultiplexor mediates access to the LCD and touch screen to prevent such attacks. It enforces the invariant that only one module may have control over the LCD and touch screen at a time: either VoteConfirmation or VoteSelection may have control, but not both. Moreover, VoteConfirmation is given precedence: if it requests control, it is given exclusive access and VoteSelection is locked out. Thus, our system can establish a trusted path between the voter interface and the VoteConfirmation module.

5.3.3 Hardware-enforced separation

Our architecture requires components to be protected from each other, so that a malicious VoteSelection component cannot tamper with or observe the state or code of other components. One possibility would be to use some form of software isolation, such as putting each component in a separate process (relying on the OS for isolation), in a separate virtual machine (relying on the VMM), or in a separate Java applet (relying on the JVM).

Instead, we use hardware isolation as a simple method for achieving strong isolation. We execute each module on its own microprocessor (with its own CPU, RAM, and I/O interfaces). This relies on physical isolation in an intuitive way: if two microprocessors are not connected by any communication channel, then they cannot directly affect each other. Verification of the interconnection topology of the components in our architecture consequently reduces to verifying the physical separation of the hardware and verifying the interconnects between them. Historically, the security community has focused primarily on software isolation because hardware isolation was viewed as prohibitively expensive [71]. However, we argue that the price of a microprocessor has fallen dramatically enough that today hardware isolation is easily affordable, and we believe the reduction in complexity easily justifies the extra cost.

With this approach to isolation, the communication elements between modules acquire special importance, because they determine the way that modules are able to interact. We carefully structured our design to simplify the connection topology as much as possible. Figure 5.2 summarizes the interconnectivity topology, and we describe several key aspects of our design below.

We remark that when multiple hardware components are used, one should ensure that the same versions of code run on each component.

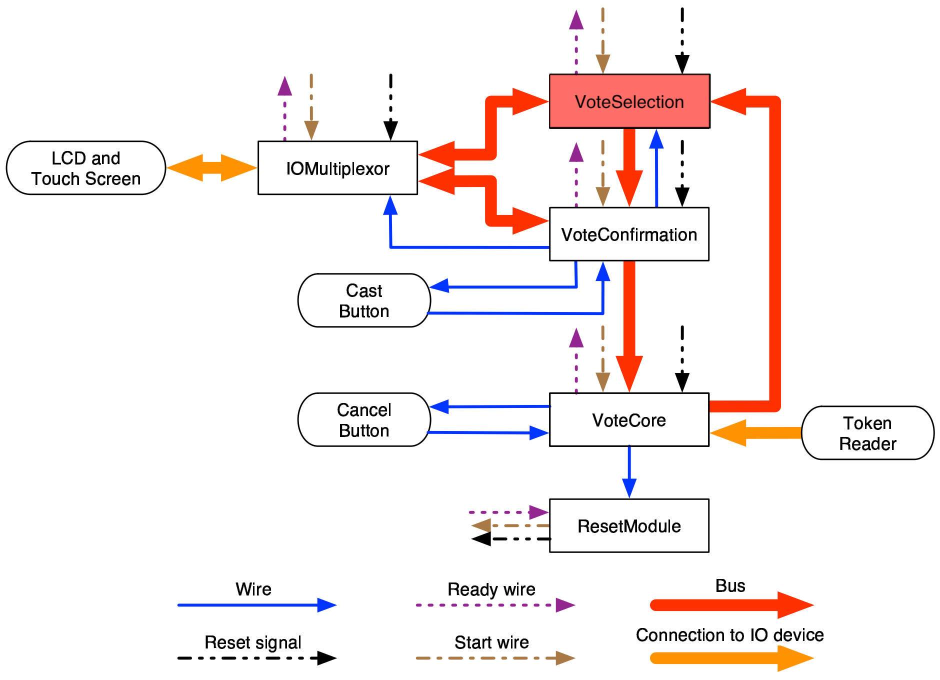


Figure 5.2: Our architecture, showing the hardware communication elements.

Buses and wires. Our hardware-based architecture employs two types of communication channels: buses and wires. Buses provide high-speed unidirectional or bidirectional communication between multiple components. Wires are a simple signaling element with one bit of state; they can be either high or low, and typically are used to indicate the presence or absence of some event. Wires are unidirectional: one component (the sender) will set the value of a wire but never read it, and the other component (the receiver) will read the value of the wire but never set it. Wires are initially low, and can be set, but not cleared; once a wire goes high, it remains high until its controlling component is reset. We assume that wires are reliable but buses are potentially unreliable.

To deal with dropped or garbled messages without introducing too much complexity, we use an extremely simple communication protocol. Our protocol is connectionless and does not contain any in-band signaling (e.g., SYN or ACK packets). When a component in our architecture wishes to transmit a message, it will repeatedly send that message over the bus until it is reset or it receives an out-of-band signal to stop transmitting. The sender appends a hash of the message to the message. The receiver accepts the first message with a valid hash, and then acknowledges receipt with an out-of-band signal. This acknowledgment might be conveyed by changing a wire's value from low to high, and the sender can poll this wire to identify when to stop transmitting. Components that need replay protection can add a sequence number to their messages.

Using buses and wires. We now describe how to instantiate the communication paths in our high-level design from Section 5.3.2 with buses and wires. Once the VoteCore module reads a valid token, it repeatedly sends the data on the token to VoteSelection until it receives a message from VoteConfirmation. After storing the vote and canceling the authentication token, the VoteCore module triggers a reset by setting its wire to the ResetModule high.

To communicate with the voter, the VoteSelection component creates a bitmap of an image, packages that image into a message , and repeatedly sends that message to the IOMultiplexor. Since the VoteSelection module may send many images, it includes in each message a sequence number; this sequence number does not change if the image does not change. Also included in the message is a list of virtual buttons, each described by a globally unique button name and the x- and y-coordinates of the region. The IOMultiplexor will continuously read from its input source (initially the VoteSelection module) and draw to the LCD every bitmap that it receives with a new sequence number. The IOMultiplexor also interprets inputs from the touch screen, determines whether the inputs correspond to a virtual button and, if so, repeatedly writes the name of the region to the

VoteSelection module until it has new voter input. Naming the regions prevents user input on one screen from being interpreted as input on a different screen.

When the voter chooses to proceed from the vote selection phase to the vote confirmation phase, the VoteConfirmation module will receive a ballot from the VoteSelection module. The VoteConfirmation module will then set its wire to the IOMultiplexor high. When the IOMultiplexor detects this wire going high, it will empty all its input and output bus buffers, reset its counter for messages from the VoteSelection module, and then only handle input and output for the VoteConfirmation module (ignoring any messages from VoteSelection). If the VoteConfirmation module determines that the user wishes to return to the VoteSelection module and edit her votes, the VoteConfirmation module will set its wire to the VoteSelection module high. The VoteSelection module will then use its bus to VoteConfirmation to repeatedly acknowledge that this wire is high. After receiving this acknowledgment, the VoteConfirmation module will reset itself, thereby clearing all internal state and also lowering its wires to the IOMultiplexor and VoteSelection modules. Upon detecting that this wire returns low, the IOMultiplexor will clear all its input and output buffers and return to handling the input and output for VoteSelection. The purpose for the handshake between the VoteConfirmation module and the VoteSelection module is to prevent the VoteConfirmation module from resetting and then immediately triggering on the receipt of the voter's previous selection (without this handshake, the VoteSelection module would continuously send the voter's previous selections, regardless of whether VoteConfirmation reset itself).

5.3.4 Reducing the complexity of trusted components

We now discuss further aspects of our design that facilitate the creation of implementations with minimal trusted code.

Resets. Each module (except for the ResetModule) interacts with the ResetModule via three wires, the initial values of which are all low: a ready wire controlled by the component and reset and start wires controlled by the ResetModule. The purpose of these three wires is to coordinate resets to avoid a situation where one component believes that it is handling the i-th voter while another component believes that it is handling the $(i+1)$-th voter.

The actual interaction between the wires is as follows. When a component first boots, it waits to complete any internal initialization steps and then sets the ready wire high. The component then blocks until its start wire goes high. After the ready wires for all components connected to the ResetModule go high, the ResetModule sets each component's start wire high, thereby allowing all components to proceed with handling the first voting session.

Upon completion of a voting session, i.e., after receiving a signal from the VoteCore component, the ResetModule sets each component's reset wire high. This step triggers each component to reset. The ResetModule keeps the reset wires high until all the component ready wires go low, meaning that the components have stopped executing. The ResetModule subsequently sets the reset wire low, allowing the components to reboot. The above process with the ready and start wires is then repeated.

Cast and cancel buttons. Our hardware architecture uses two physical buttons, a cast button and a cancel button. These buttons directly connect the user to an individual component, simplifying the task of establishing a trusted path for cast and cancel requests. Our use of a hardware button (rather than a user interface element displayed on the LCD) is intended to give voters a way to know that their vote will be cast. If we used a virtual cast button, a malicious VoteSelection module could draw a spoofed cast button on the LCD and swallow the user's vote, making the voter think that they have cast their vote when in fact nothing was recorded and leaving the voter with no way to detect this attack. In contrast, a physical cast button allows attentive voters to detect these attacks (an alternative might be to use a physical "vote recorded" light in the VoteCore). Additionally, if we used a virtual cast button, miscalibration of the touch screen could trigger accidental invocation of the virtual cast button against the voter's wishes. While calibration issues may still affect the ability of a user to scroll through a multi-screen confirmation process, we anticipate that such a problem will be easier to recover from than touch screen miscalibrations causing the DRE to incorrectly store a vote. To ensure that a malicious VoteSelection module does not trick the user into pressing the cast button prematurely, the VoteConfirmation module will only enable the cast button after it detects that the user paged through all the vote confirmation screens.

We want voters to be able to cancel the voting process at any time, regardless of whether they are interacting with the VoteSelection or VoteConfirmation modules. Since the VoteSelection module is untrusted, one possibility would be to have the IOMultiplexor implement a virtual cancel button or conditionally pass data to the VoteConfirmation module even when the VoteSelection module is active. Rather than introduce these complexities, we chose to have the VoteCore module handle cancellation via a physical cancel button. The cancel button is enabled (and physically lit by an internal light) until the VoteCore begins the process of storing a ballot and canceling an authentication token.

5.4 Prototype implementation

To evaluate the feasibility of the architecture presented in Section 5.3, we built a prototype implementation. Our prototype uses off-the-shelf "gumstix connex 400xm" computers. These

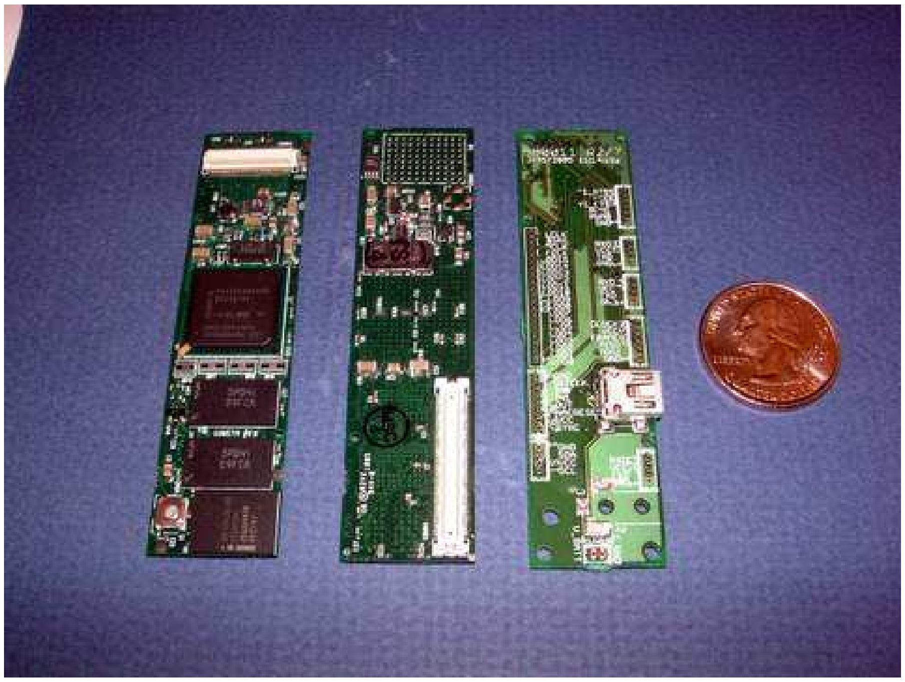


Figure 5.3: We show the front and back of a gumstix as well as an expansion board through which the GPIO and serial ports are soldered. The quarter gives an indication of the physical size of these components.

---

computers measure 2cm by 8cm in size, cost $144 apiece, and contain an Intel XScale PXA255 processor with a 400 MHz StrongARM core, 64 MB of RAM, and 16 MB of flash for program storage. We enable hardware isolation by using a separate gumstix for each component in our architecture.

We do not claim that the gumstix would be the best way to engineer an actual voting system intended for use in the field. However, the gumstix have many advantages as a platform for prototyping the architecture. In conjunction with an equally sized expansion board, the processors support three external RS-232 serial ports, which transmit bidirectional data at 115200 kbps. We use serial ports as our buses. Additionally, each gumstix supports many general purpose input/output (GPIO) registers, which we use for our wires. Finally, the XScale processor supports an LCD and

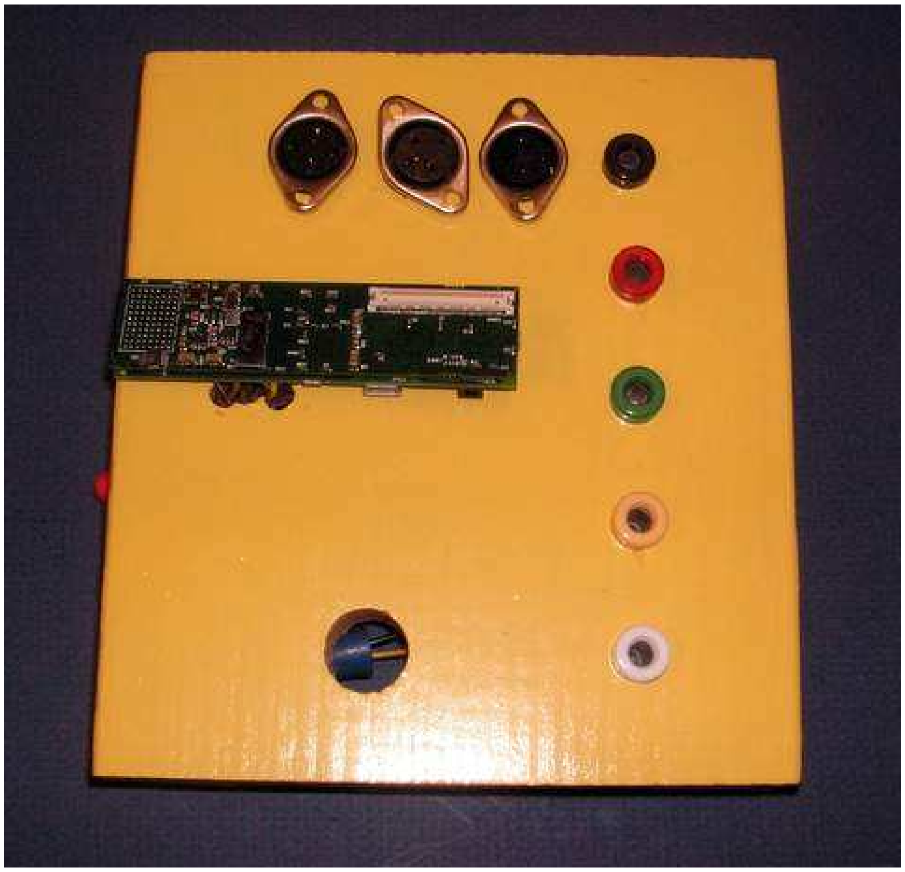


Figure 5.4: The mounting board for a single component. It contains three serial ports (along the top), 4 GPIO pins and a ground pin (along the right side), as well as a gumstix processor board mounted atop an expansion board.

---

touch screen interface.

The gumstix platform's well-designed toolchain and software environment greatly simplified building our prototype. The gumstix, and our prototype, use a minimal Linux distribution as their operating system. Our components are written in Java and run on the Microdoc J9 Java VM; its JIT provides a significant speed advantage over the more portable JamVM Java interpreter. Our choice of Java is twofold: it is a type-safe language and so prevents a broad range of exploits; secondly, several program verification tools are available for verifying invariants in Java code [15, 44]. C# is another natural language choice since it too is type-safe and the Spec# [7] tool could aid in verification, but C# is not supported as well on Linux. We view a rich stable of effective verification tools to be just as important as type-safety in choosing the implementation language since software tools can improve confidence in the voting software's correctness. Both can eliminate large classes of bugs.

5.4.1 Implementation primitives

Our architecture requires implementations of two separate communications primitives: buses and wires. It is straightforward to implement buses using serial ports on the gumstix. To do so, we expose connectors for the serial ports via an expansion board connected to the main processor. varch/Figures 5.3 and 5.4 show an example of such an expansion board. We additionally disable the `getty` terminal running on the serial ports to allow conflict free use of all three serial ports. The PXA255 processor has 84 GPIO pins, each controlled by registers; we implement wires using these GPIOs. A few of the pins are exposed on our expansion board and allow two components to be interconnected via their exposed GPIO pins. Each GPIO pin can be set in a number of modes. The processor can set the pin "high" so that the pin has a 3.3 volt difference between the reference ground; otherwise, it is low and has a 0 voltage difference between ground. Alternatively, a processor can poll the pin's state. To enforce the unidirectional communication property, particularly when a single wire is connected to more than two GPIOs, we could use a diode, which allows current to flow in only one direction [3]. We currently rely on software to enforce that once a GPIO is set high, it cannot ever be set low without first restarting the process; this is a property one could

[3]Even this may not be enough, since an actual diode does not behave as the idealized diode we rely upon.

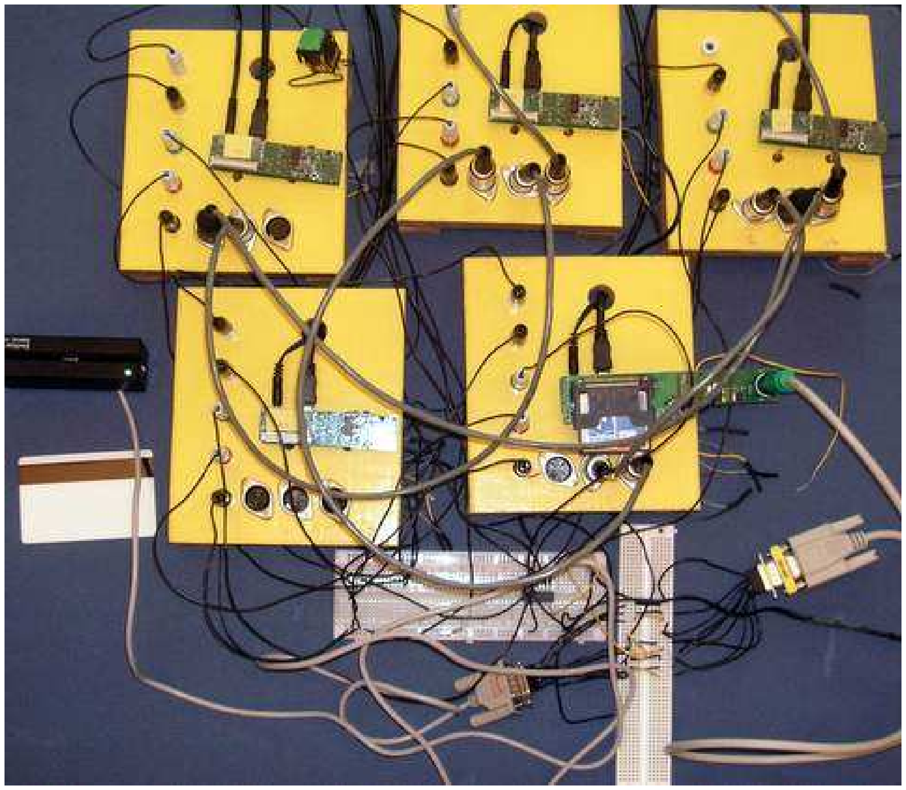


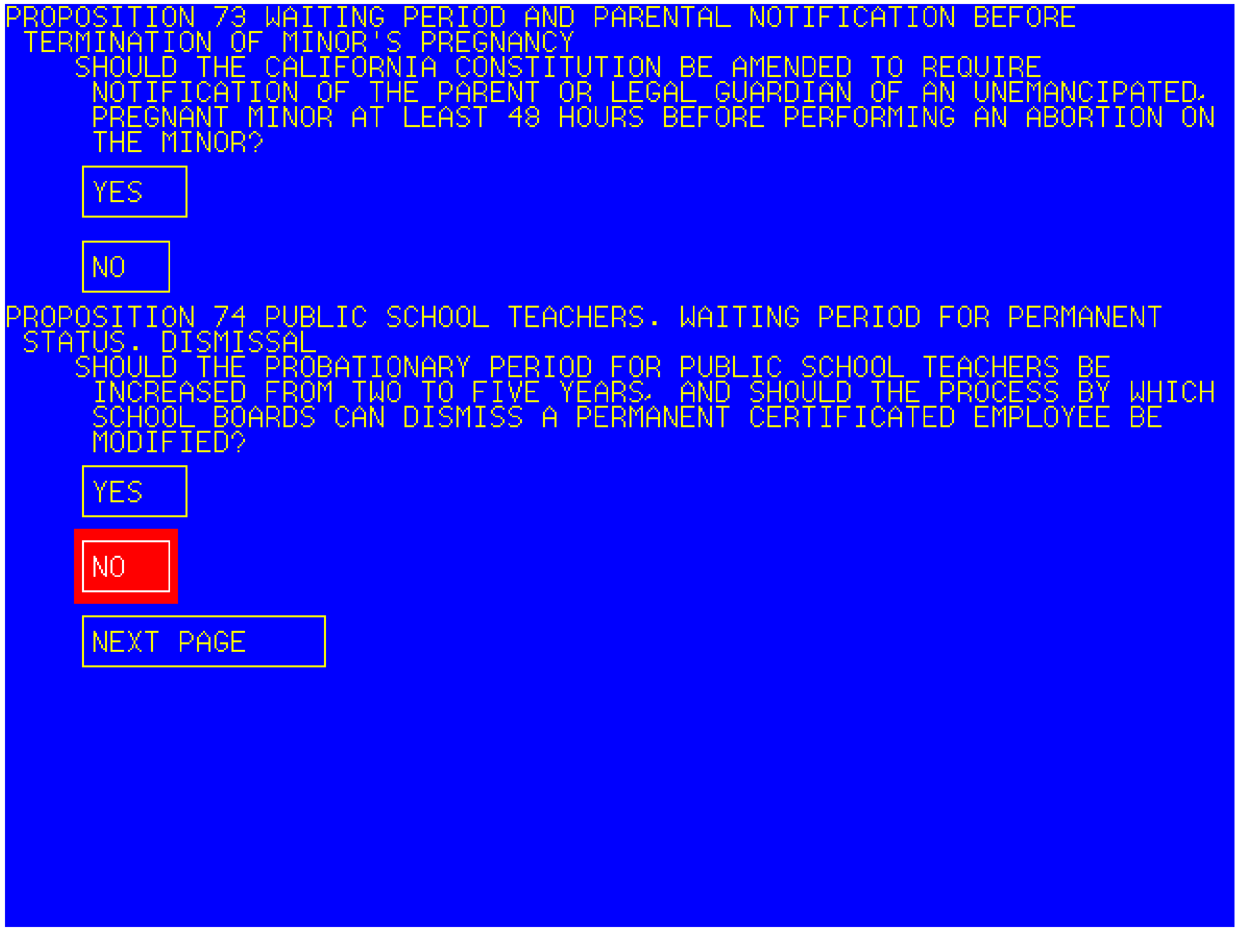
Figure 5.5: A picture of our prototype implementation. There is one board for each component in the system. The magnetic swipe card (along the left) is used for authentication, while the cast button is in the upper left component.

enforce in hardware via a latch, though our current prototype does not do so yet.

In addition to the GPIOs, the PXA255 exposes an NRESET pin. Applying a 3.3v signal to the NRESET pin causes the processor to immediately halt execution; when the signal is removed, the processor begins in a hard boot sequence. The gumstix are able to reboot in under 10 seconds without any optimizations, making the NRESET pin nearly ideal to clear a component's state during a reset. Unfortunately, the specifics of the reboot sequence causes slight problems for our usage. While the NRESET wire is held high, the GPIO pins are also high. In the case where one component reboots before another (or where selective components are reboot), setting the GPIOs high will inadvertently propagate a signal along the wire to the other components. Ideally, the pins would be low during reset. We surmise that designing a chip for our ideal reset behavior would not be difficult given sufficient hardware expertise. Since the microprocessors in our platform do not exhibit our ideal behavior, in our prototype we have a separate daemon connected to an ordinary GPIO wire that stops the Java process running the component code when the reset pin goes high and then resets all wire state to low. The daemon starts a new component process when the signal to its reset pin is removed. This is just a way of emulating, in software, the NRESET semantics we prefer. Of course, a production-quality implementation would enforce these semantics in trusted hardware.

We use a Kanecal KaneSwipe GIT-100 magnetic card reader for authorizing voters to use the machine. A voter would receive a card with authentication information on it from poll workers upon signing in. The voter cannot forge the authentication information (since it contains a public key signature), but can use it to vote once on a designated DRE. The reader has an RS-232 interface, so we are able to use it in conjunction with the serial port on the gumstix.

Finally, our implementation of the VoteCore component uses a compact flash card to store cast ballot images and invalid magcard identifiers. Election officials can remove the flash card and transport it to county headquarters after the close of polls. A deployed DRE might use stronger privacy-protection mechanisms, such as a history-independent, tamper-evident, and subliminal-free data structure [55]. For redundancy, we expect a deployed DRE to also store multiple copies of the votes on several storage devices. A full implementation of the VoteSelection component would likely also use some kind of removable storage device to store the ballot definition file. In our prototype, we hard-code a sample ballot definition file into the VoteSelection component. This suffices for our purposes in gauging the feasibility of other techniques.



PROPOSITION 73 WAITING PERIOD AND PARENTAL NOTIFICATION BEFORE TERMINATION OF MINOR'S PREGNANCY

SHOULD THE CALIFORNIA CONSTITUTION BE AMENDED TO REQUIRE NOTIFICATION OF THE PARENT OR LEGAL GUARDIAN OF AN UNEMANCIPATED, PREGNANT MINOR AT LEAST 48 HOURS BEFORE PERFORMING AN ABORTION ON THE MINOR?

YES

NO

PROPOSITION 74 PUBLIC SCHOOL TEACHERS. WAITING PERIOD FOR PERMANENT STATUS. DISMISSAL

SHOULD THE PROBATIONARY PERIOD FOR PUBLIC SCHOOL TEACHERS BE INCREASED FROM TWO TO FIVE YEARS, AND SHOULD THE PROCESS BY WHICH SCHOOL BOARDS CAN DISMISS A PERMANENT CERTIFICATED EMPLOYEE BE MODIFIED?

YES

[✓ NO]

NEXT PAGE

Figure 5.6: The image shows a screenshot of the VoteSelection component displaying referenda from the November 2005 election in Berkeley, CA. We flipped a coin to choose the response shown on this screen.

Our prototype consists of five component boards wired together in accordance with Figure 5.2. We implement all of the functionality except for the cancel button. See Figure 5.5 for a picture showing the five components and all of their interconnections. Communication uses physical buses and wires. The I/O multiplexer, after each update operation, sends an image over a virtual bus connected (connected via the USB network) to the PC for I/O. It sends the compressed image it would ordinarily blit to the framebuffer to the PC so that the PC can blit it to its display. The gumstix only recently supported LCD displays, and we view our PC display as an interim solution. The additional software complexity for using the LCD is minimal as it only requires blitting an image to memory.

Figure 5.6 shows our voting software running on the gumstix. We used ballot data from the November 2005 election in Alameda County, California.

5.5 Evaluation

5.5.1 Verifying the desired properties

Property 1. Recall that to achieve "memorylessness" we must be able to show the DRE is always reset after a voter has finished using the machine, and the DRE only opens a given file read-only or write-only, but not both. To show that the DRE is reset after storing a vote, we examine a snippet of the source code from `VoteCore.java`, the source code for the VoteCore module in Figure 5.7. In line 7, after storing the ballot into the ballot box, the VoteCore module continuously raises the reset wire high. Looking at the connection diagram from Figure 5.2, we note the reset wire terminates at the ResetModule and induces it to restart all components in the system. Further inspecting code not reproduced in Figure 5.7 reveals the only reference to the `ballotbox` is in the constructor and in

```java
grabio.set();
...  UPDATE DISPLAY ...
castenable.set();
if (cast.isSet()) {
   while (true) {
      toVoteCore.write(ballot);
   }
}
```

Confirm.java

```java
byte [] ballot =
       fromVoteConf.read();
if (ballot != null) {
   ...  INVALIDATE VOTER TOKEN ...
   ballotbox.write (ballot);
   while (true) {
       resetWire.set();
   }
}
```

VoteCore.java

Figure 5.7: Code extracts from the VoteConfirmation and VoteCore modules, respectively. Examining these code snippets with the connection topology helps us gain assurance that the architecture achieves Properties 1 and 2.

line 5, so writes to it are confined to line 5.

Finally, we need merely examine every file open call to make sure they are either read-only or write only. In practice, we can guarantee this by ensuring writable files are append-only, or for more sophisticated vote storage mechanisms as proposed by Molnar et al., that the storage layer presents a write-only interface to the rest of the DRE.

Property 2. For the "consent-to-cast" property, we need to verify two things: 1) the ballot can only enter the VoteCore through the VoteConfirmation module, and 2) the voter's consent is required before the ballot can leave the VoteConfirmation module.

Looking first at `Confirm.java` in Figure 5.7, the VoteConfirmation module first ensures it has control of the touch screen as it signals the IOMultiplexor with the "grabio" wire. It then displays the ballot over the bus, and subsequently enables the cast button. Examining the hardware will show the only way the wire can be enabled is through a specific GPIO, in fact the one controlled by the "castenable" wire. No other component in the system can enable the cast button, since it is not connected to any other module. Similarly, no other component in the system can send a ballot to the VoteCore module: on line 6 of `Confirm.java`, the VoteConfirmation sends the ballot on a bus named "toVoteCore", which is called the "fromVoteConf" bus in `VoteCore.java`. The ballot is demarshalled on line 1. Physically examining the hardware configuration confirms these connections, and shows the ballot data structure can only come from the VoteConfirmation module. Finally, in the VoteCore module, we see the only use of the ballotbox is at line 5 where the ballot is written to the box. There are only two references to the BallotBox in the `VoteCore.java` source file (full file not shown here), one at the constructor site and the one shown here. Thus we can be confident that the only way for a ballot to be passed to the BallotBox is if a voter presses the cast

	Java	**C (JNI)**	**Total**
Communications	2314	677	2991
Display	416	52	468
Misc. (interfaces)	25	0	25
VoteSelection	377	0	377
VoteConfirmation	126	0	126
IOMultiplexor	77	0	77
VoteCore	846	54	900
ResetModule	121	0	121
Total	4302	783	5085

Table 5.1: Non-comment, non-whitespace lines of code.

button, indicating their consent. We must also verify that the images displayed to the voter reflect the contents of the ballot.

5.5.2 Line counts

One of our main metrics of success is the size of the trusted computing base in our implementation. Our code contains shared libraries (for communications, display, or interfaces) as well as each of the main four modules in the TCB (VoteConfirmation, IOMultiplexor, VoteCore, and ResetModule). The VoteSelection module can be excluded from the TCB when considering Properties 1 and 2. Also included in the TCB, but not our line count figures, are standard libraries, operating system code, and JVM code.

In Table 5.1, we show the size of each trusted portion as a count of the number of source lines of code, excluding comments and whitespace.

The communications libraries marshal and unmarshal data structures and abstract the serial devices and GPIO pins. The display libraries render text into our user interface (used by the

VoteConfirmation component) and ultimately to the framebuffer.

5.6 Applications to VVPATs and cryptographic voting protocols

So far we've been considering our architecture in the context of a stand-alone paperless DRE machine. However, jurisdictions such as California require DREs to be augmented with a voter verified paper audit trail. In a VVPAT system, the voter is given a chance to inspect the paper audit trail and approve or reject the printed VVPAT record. The paper record, which remains behind glass to prevent tampering, is stored for later recounts or audits.

VVPAT-enabled DREs greatly improve integrity protection for non-visually impaired voters. However, a VVPAT does not solve all problems. Visually impaired voters who use the audio interface have no way to visually verify the selections printed on the paper record, and thus receive little benefit from a VVPAT. Also, a VVPAT is only an integrity mechanism and does not help with vote privacy. A paper audit trail cannot prevent a malicious DRE from leaking one voter's choices to the next voter, to a poll worker, or to some other conspirator. Third, VVPAT systems require careful procedural controls over the chain of custody of paper ballots. Finally, a VVPAT is a fall-back, and even in machines that provide a VVPAT, one still would prefer the software to be as trustworthy as possible.

For these reasons, we view VVPAT as addressing some, but not all problems. Our methods can be used to ameliorate some of the remaining limitations, by providing better integrity protection for visually impaired voters, better privacy protection for all voters, reducing the reliance on procedures for handling paper, and reducing the costs of auditing the source code. Combining our methods with a VVPAT would be straightforward: the VoteConfirmation module could be augmented with support for a printer, and could print the voter's selections at the same time as they are displayed on the confirmation screen. While our architecture might be most relevant to jurisdictions that have decided, for whatever reason, to use paperless DREs, we expect that our methods could offer some benefits to VVPAT-enabled DREs, too.

Others have proposed cryptographic voting protocols to enhance the security of DREs [19, 39, 58, 60]. We note that our methods could be easily combined with those cryptographic schemes.

5.7 Extensions and discussion

Other properties. In this chapter, we have extensively discussed verifying Properties 1 and 2; Work done in conjunction with Molnar et al. addresses Property 3, and Chapter 6 describes a software checker for Property 6. That leaves Property 4 (DRE only stores ballots the voter approves) and Property 5 (Ballot contains nothing more than the voter's choices).

Briefly, we sketch how the architecture presented in this chapter aids verification of both these properties. For Property 4, it is easy to inspect, or write a software checker, that the ballot is unmodified after the VoteConfirmation module. The same ballot that enters the module is the one that is sent to the BallotBox. Given the (linear) dataflow path in the architecture, it is easy to verify the ballot data is unmodified. Of course, this leaves verifying that the display routines in the VoteConfirmation accurately reflect the ballot data for confirmation.

Finally, for Property 5, we can use a check function to guarantee that the ballot is in a canonical format. Given that Property 4 guarantees that the ballot remains unmodified, a (boolean) canonicalization function run during execution can leverage the fail-stop mode and halt on non-canonical ballots. Chapter 6 addresses the benefits of the fail-stop model for verifying properties.

Minimizing the underlying software platform. Our prototype runs under an embedded Linux distribution that is custom designed for the gumstix platform. Despite its relatively minimal size (4MB binary for kernel and all programs and data), it still presents a large TCB, most of which is unnecessary for a special-purpose voting appliance. We expect that a serious deployment would dispense with the OS and build a single-purpose embedded application running directly on the hardware. For instance, we would not need virtual memory, memory protection, process scheduling, filesystems, dual-mode operation, or most of the other features found in general-purpose operating systems. It might suffice to have a simple bootloader and a thin device driver layer specialized to just those devices that will be used during an election. Alternatively, it may be possible to use ideas from nanokernels [23], microkernels [29, 69], and operating system specialization [68] to reduce the operating system and accordingly the TCB size.

Deploying code. Even after guaranteeing the software is free of vulnerabilities, we must also guarantee that the image running on the components is the correct image. This is not an easy problem, but the research community has begun to address the challenges. SWATT [80] is designed to validate the code image on embedded platforms, though their model does not allow for CPUs with virtual memory, for example. TCG and NGSCB use a secure hardware co-processor to achieve the same ends, though deploying signed and untampered code to devices still requires much work. Additionally, a human must then check that all components are running the latest binary and must ensure that the binaries are compatible with each other – so that a version 1.0 VoteCore is not running with a version 1.1 IOMultiplexor module, for example.

This concern is orthogonal to ours, as even current voting machines must deal with versioning. It illustrates one more challenge in deploying a secure voting system.

5.8 Conclusions

Our approach uses hardware to isolate components from each other and uses reboots to guarantee voter privacy. In particular, we have shown how isolating the VoteSelection module, where much of the hairiness of a voting system resides, into its own module can eliminate a great deal of complex code from the TCB. Though isolation is not a novel idea, the way we use it to improve the security of DREs is new. This work shows that it is possible to improve existing DREs without modifying the existing voter experience or burdening the voter with additional checks or procedures.

The principles and techniques outlined here show that there is a better way to design voting systems.

Chapter 6

Environment-freeness

In this chapter, we seek to develop software analysis techniques that guarantee that the in-memory copy of the ballot can be properly recovered after serialization for later tallying. To do so, we introduce the notion of *environment-free* functions, where the function's behavior depends only and deterministically on the arguments to the function. Then, we show to use this concept to verify the correct invertability of Encode operations such as serialization, compression, and encryption through a mixture of static analysis and runtime checks. The strategy is to first verify that the Decode implementation is environment-free and then add a simple runtime check to ensure that the encoded data can and will be correctly decoded in the future. We develop a static analysis for verifying that Java code is environment-free. To demonstrate its feasibility, we implemented our algorithm as an Eclipse plug-in and used it to analyze the serialization routines in our voting architecture from Chapter 3 and also to verify that decryption is the inverse of encryption in a Java cryptography implementation.

Parts of this work are drawn with permission from prior work [75].

6.1 Introduction and motivation

Many computer programs perform serialization and deserialization, converting an in-memory version of a data structure into a form suitable for storage or transmission and back again. In this chapter, we develop novel methods for verifying the correctness of serialization and deserialization code. In particular, we wish to verify that deserialization is the inverse of serialization, i.e., that serializing a data structure and then deserializing the result will give you back the same data structure you started with.

Verifying the correctness of serialization and deserialization is a difficult task. Serialization and deserialization typically involve walking a (potentially cyclic) object graph, and thus inevitably implicate complex aliasing issues. Reasoning about aliasing is well known to be challenging. Also, the invariants needed to prove the correctness of serialization and deserialization may not be immediately apparent from the code and may be messy and unilluminating when written down explicitly. Therefore, standard formal methods appear to be ill-suited for this task.

More broadly, serialization is just one of a family of common data transformation routines that litter voting software. Two others in the family include encryption/decryption and compression/decompression.

We seek to verify the following property about a pair of algorithms, $(\mathsf{Encode}, \mathsf{Decode})$: namely, for all x, $\mathsf{Decode}(\mathsf{Encode}(x))$ should yield some output x' that is functionally equivalent to x. We want this property to hold even if Decode is invoked at some later time on some other machine, so we will also need to verify that Decode does not implicitly depend on any data (other than its input) that might be different on some other machine. We call this the Inverse Property, since the goal is to verify that Decode is a left inverse of Encode. In many contexts, it is a serious error if Decode fails to yield the original input.

We use one specific aspect of voting machine accuracy as a running example in this paper. As the voter makes selections, the voting machine accumulates these selections into a data structure in RAM. When the voter casts her ballot, the machine must serialize (Encode) this data structure to disk. During the tallying stage, the disk will be read, and the choices will need to be deserialized (Decode) into the voter's original data structure in order to compute the tally. We wish to verify that the vote data structure that is serialized and recorded to disk when the voter casts her ballot can later be reconstructed exactly as it was when the voter cast her ballot. A failure to reconstruct the original data structure would be a serious problem, because it would mean that a voter's choices could not be recovered accurately, disenfranchising the voter.

6.2 Static analysis to enable dynamic checking

Statically analyzing the correctness of a pair of algorithms to verify that the second is *always* the inverse of the first is beyond our expertise. It is easier to support *fail-stop* operation, in which errors are detected at runtime but before any harmful consequences have taken place. The current transaction leading to the error is then cancelled (or possibly retried, if the error is likely to be transitory).

Returning to our example, a voting machine endowed with this mechanism would verify the Inverse Property for each voter's ballot before announcing to that voter that their vote was successfully cast. If the check fails, the voter would be notified and advised to use another voting machine. Without the check, the voter would never know that their ballot had been improperly serialized (and hence stored); depending upon the nature of the deserialization error, the problem may or may not be caught at tally time when their vote is counted.

Note that checking the Inverse Property requires knowledge about a hypothetical future; to confirm a voter's vote we must be confident that any future attempt to deserialize their ballot will be successful. Ensuring this requires us to be able to predict the future behavior of the Decode method. The easiest way to make such a method predictable is to require it to "always do the same thing" and to check its behavior once, with a check like the following:

```
$y := \mathsf{Encode}(x)$
abort if $x \neq \mathsf{Decode}(y)$
```

For instance, in the voting machine example, we would translate the pseudo-code above into a concrete Java implementation as follows:

```
byte[] bytes = ballot.serialize();
assert(ballot.equals(
         Ballot.deserialize(bytes)));
```

The runtime assertion check is intended to ensure that the serialized `bytes` will properly deserialize into the `ballot`. By checking that the deserialization is correct at the time of serialization, we'd like to then infer that deserialization will be correct at some *later* time, when the `deserialize()` function (or more generally the Decode function) will be run. However, this inference is only valid if we make several assumptions about the behavior of the `deserialize()` and `equals()` methods.

1. The result of the `deserialize()` function must be a deterministic function of its arguments, namely `bytes`. Its output must not depend upon any other values, such as the values of global variables, the time of day, or the contents of the filesystem. The `deserialize()`

function must yield the same results when it is later run on the same input, even if it is run on another machine at a later time.

2. The `deserialize()` function must not be able to modify global state; i.e. it can only modify objects reachable from its arguments [1].

3. The `equals()` method must check all relevant properties of the `ballot` object and does not have any side-effects. We will take it as the specification of what it means for two `ballot` objects to be functionally equivalent.

4. The `deserialize()` function that will be executed later (including any methods or static declarations it makes use of) must be the same one used in the runtime check.

If we can statically verify that these four requirements are met, then we will be entitled to conclude that the serialized data will later be deserialized correctly.

Note that we have explicitly not restricted the serialization function in any way. For example, we don't require the Encode function to be deterministic. In general, Encode might depend on a source of randomness or non-determinism in generating its output. This is particularly important for encryption functions. As long as the Decode function deterministically reconstructs the original data, it does not matter how it operates in any way. For example, we don't require the `serialize()` function to be deterministic. In the general case, Encode should be able to depend on a source of non-determinism in generating its output. This is particularly important for encryption functions. As long as the Decode function deterministically reconstructs its input, it does not matter how the Encode function works.

[1] If the `deserialize()` function is passed a new deep copy of any arguments that it may mutate, the `assert()` statement does not change the behavior of the program if it succeeds. In our case, making a deep copy of a `byte[]` is trivial.

In summary, our strategy is as follows. First, we transform the code by introducing a run-time assertion check after every call to Encode. For arguments that are mutated by the Decode function, we pass it deep copies instead of the originals. Second, we manually confirm that the third and fourth requirements are met. Finally, we use static analysis to verify that that the first two requirements are met. This strategy suffices to ensure that the program satisfies fail-stop correctness: if the transformed program does not abort, then the Inverse Property will be satisfied on that execution.

This paper addresses the first two of the above requirements; we develop a static analysis to make sure that the Decode function computes its output deterministically based only on its input and does not cause disruptive side effects. Our static analysis is designed to place as few restrictions on the rest of the code as possible.

6.3 Environment-free and compile-time constants

6.3.1 Overview

One possible method to enable the fail-stop approach outlined in Section 6.2 is to require the Decode function be *pure*. A pure function is required to be free of side-effects; executing such a function and discarding the result should be a no-op. Depending on whose definition one uses, a pure function may or may not be allowed to read the values of potentially mutable global state; JML seems to allow it [73] as it does not violate the no-op-equivalence requirement.

Pureness, at least in the JML sense, is thus both overly restrictive and not restrictive enough for our purposes. We do not require the Decode function be side-effect free in general, but we do restrict its side effects to objects reachable from its arguments. In-place array manipulations are common in decryption algorithms, and we wish to be able to support this pattern. Making a deep copy of arguments before calling the checker suffices to ensure that the check won't modify them.

Allowing pure functions to read static state is problematic, as purity is only adequate to ensure that two executions of Decode on equivalent inputs yield the same result when global variables are the same. Equivalence of global states is not an easy property to check.

Instead, we propose eliminating the ability for the Decode function to read from global constants that may vary. We rely on a new concept, called *environment-freeness*, to describe the property that the Decode function should exhibit.

6.3.2 Environment-free functions

A function is environment-free if it satisfies the following two restrictions:

1. It may not cause any externally-visible mutation except to modify objects reachable from its arguments. This includes modifications of global state and state external to the program, e.g. disk or network.

2. Any two calls to that function with equal arguments are guaranteed to return (or throw) equal results. This sense of equality is defined below. For instance methods in Java, the implied `this` parameter is considered an argument.

For the purposes of this definition, equality between two Java objects is defined as equivalence of the graphs of objects reachable from the two objects. Equality over these graphs includes the values of all primitive fields as well as the structure of pointers in the graph (including aliasing of objects reachable from compile-time constants). There is a special exception for the stack traces stored in `Throwable` objects, as will be explained in Section 6.4.4. This reflects the strongest notion of equality that is observable by an environment-free method, since the addresses of objects are not visible to such methods, as discussed in Section 6.5.3. It is also the relationship preserved by default by Java serialization.

Note that this definition allows an environment-free function to modify objects that are reachable from its arguments. Changing these arguments may cause different versions of the same Java object to be unequal given the definition of equality above.

6.3.3 Compile-time constants

A *compile-time constant* is either a lexical constant that appears in the source code or a global variable that is guaranteed to have the same value at all times. The value of a compile-time constant must be a deterministic function of the program source code and must be the same on every execution of the program.

This notion is useful in our static analysis for verifying that a function is environment-free. For a function to be a strictly deterministic function of its arguments, it cannot depend on any other values that may vary. It would suffice to exclude access to all global (static) fields, but we need not be that strict. It is safe to access a static field provided that it is guaranteed to have a deterministic, constant value. If we can be certain that every time the field is accessed it always has the same value, it is possible to guarantee that the environment-free function will return the same value when given the same input even if called at different times. Therefore, our static analysis algorithm imposes the requirement that any global variable that is accessed by an environment-free function must be a compile-time constant.

6.3.4 How these are verified

At a high level, we can verify that a function is environment-free by confirming that it does not read any external state, apart from what is reachable via its arguments or via compile-time constants. An environment-free function may call other functions, but we require those other functions to be environment-free, too. Permitting an environment-free function `f()` to call a non–environment-free function could potentially allow a source of non-determinism to affect `f()`'s return value, breaking the guarantee of determinism.

We verify that a global variable is a compile-time constant by checking that three conditions are met. Firstly, its value must be set before any environment-free method that might access it is invoked; and secondly, its value must not be changed for the life of the program. These conditions are verified using an extended notion of the checks that the Java compiler uses to enforce the `final` attribute of fields. Finally, the global variable must be initialized to a deterministic value; i.e., its initialization expression must be a deterministic function of the program source code. The restrictions on the initializer are similar to those on environment-free fields. The initializer is only allowed to rely on other (already initialized) compile-time constants and may only call environment-free methods.

6.4 Specifics and algorithm

Here we describe the mechanisms we use to verify these properties of Java 1.4 source code.

6.4.1 Annotations

In our system, the programmer annotates each environment-free method's declaration with an annotation `@envfree` to indicate that it (and all methods it transitively calls) are alleged to be environment-free. The annotation serves as a directive to the static checker to validate the correctness of this assertion. Therefore, to verify that a `Decode` method is environment-free, all a programmer need do is mark it with an `@envfree` annotation and run our static checker.

In practice, we have found it convenient to re-factor our code slightly by introducing a `static` method that returns a boolean indicating whether its first argument deserializes into the second. In the example of Section 6.2, we would introduce the following function:

```
/** Returns whether the deserialized first
 * parameter "equals()" the second parameter.
 * @envfree */
public static boolean check(byte[] bytes,
                             Ballot b) {
  return b.equals(Ballot.deserialize(bytes));
}
```

The function would be invoked from an assertion, like this:

```
  assert(check(bytes, ballot));
```

Introducing the `check()` function allows us to reduce the number of annotation sites and clearly indicates what check is being done. The annotation on the check method will force the checker to validate that both the `equals()` method and `deserialize()` method are environment-free without extra annotations on either method. We can thus limit the programmer's burden.

We also allow a static field to be explicitly tagged with an annotation `@ctc` to indicate they are compile-time constants. However, this annotation is not strictly necessary, since all fields required to be compile-time constants will automatically be discovered. This facility may be helpful for a programmer who wishes to verify that a static field is a compile-time constant independently of its use by an environment-free method.

6.4.2 Finding methods and variables to check

The checker analyzes the source to create a list of all “root methods” annotated with the `@envfree` annotation. It then finds the transitive closure of all methods called from these root methods; all methods that are *potentially* reachable from an annotated environment-free method must also be environment-free. The checker considers any method referenced in an environment-free method as a method that could be called and hence should be checked to see whether it too is environment-free. Methods in any subclasses that override the environment-free method are also added to the list to be checked, since they too must be environment free. Added to this list are constructors that are invoked with the `new` operator within environment-free methods. These too must be environment-free since a non–environment-free constructor could introduce a source of non-determinism into what should be a deterministic environment-free function. All of these transitively reachable methods are treated as though they had been marked with an (implicit) `@envfree` annotation.

Finally, any global (`static`) variable that is annotated `@ctc` or referenced within an environment-free method must be a compile-time constant. Its declaration site, with a reference to its initializer, is queued into a list. Every element in the queue will be checked to confirm that it is indeed a compile-time constant.

We next describe the checker’s steps to check whether the list of environment-free methods and compile-time constants satisfy their requirements.

6.4.3 Compile time constants

As outlined in Section 6.3.4, compile-time constants are required to meet three requirements. We ensure the first, that its value be set before it is read by any environment-free method, by requiring it to be initialized where it is declared. The second concerns mutability: A compile-time constant field must be declared as `final`, so that the object the field points to cannot change. In addition, the field must be of a type that is immutable as explained below, to ensure that the constant object cannot be modified after it is initialized. The third and final requirement concerns determinism of initialization and is discussed in Section 6.4.3.

Immutability

A compile-time constant must have the same value over the lifetime of the program's execution. Joe-E, an object-capability subset of the Java language, provides useful definitions and implementations for immutability. For compile-time constants, we follow Joe-E's lead and require that the compile-time constant's type must implement an `Immutable` interface [53]. The interface does not have any members, but serves as a marker to the checker indicating that the class (and all its subclasses) must be transitively immutable (i.e. it must not be possible to mutate the object or any other object reachable by following its fields).

Joe-E contains a more general set of requirements for immutability of an object than are necessary for the definition of compile-time constants [2]. Our checker for this application verifies the following simplified set of properties that suffice to ensure immutability.

- All of the instance fields in an immutable type must must be declared `final`. This ensures

[2]Specifically, it allows more flexibility in the use of nested classes and superclasses that are not Immutable. For details, see the Joe-E specification [53].

that the references assigned at the type's construction will not change at some later time.

- The static type of all instance fields within of an immutable type must also implement the Immutable interface. The requirement that all instance fields must be final is not sufficient to guarantee immutability. If an immutable type *T* has an instance field that has a mutable member, then *T* is no longer immutable, due to the transitive definition of immutability. To eliminate such cases, all objects reachable from an object of type *T* also be of immutable type.

- We require all that the superclass of an immutable type *T* also implement the `Immutable` interface (with the exception of `java.lang.Object`, which has no fields). This ensures that fields in *T*'s superclass will also be Immutable, which is necessary as they are a part of *T*.

We restrict implementation of the Immutable interface to top-level classes and static inner classes. Non-static inner classes are more complicated as they are granted access to fields of their enclosing classes. It would not be difficult to support such structures, but we did not see much need for this particular programmatic pattern. Static inner classes are permitted without restriction since they are not constructed with the implicit pointer to their enclosing class. We also forbid local or anonymous classes from implementing the Immutable interface; these are more complicated as they can also inherit access to local variables in scope. While one can also create criteria for Immutable local and anonymous classes, we similarly did not see a need to support it. Our use of immutability is to support compile-time constants, which does not require classes to be defined within a method's scope.

Library types	**Primitive types**
`java.lang.Boolean`	`boolean`
`java.lang.Byte`	`byte`
`java.lang.Char`	`char`
`java.lang.Double`	`double`
`java.lang.Float`	`float`
`java.lang.Integer`	`int`
`java.lang.Long`	`long`
`java.lang.Short`	`short`
`java.lang.String`	

Table 6.1: List of types that are on the Immutable whitelist. We analyzed these library and built-in types to guarantee they honor the Immutable properties.

Primitives and whitelists

Without amending our previous rules for immutability, an immutable type could never contain primitive data types. But a final field containing a primitive type satisfies our requirements for immutability: its value cannot change once initialized. Hence, we endow the checker to accept all of Java's primitive types (Table 6.1) to be immutable, as if they implement the Immutable interface. This allows an immutable type to contain a `final int` field, for example.

Additionally, we have manually analyzed the semantics of the corresponding Object type for each primitive to ensure that they too satisfy the immutability properties (See Table 6.1). We should note that not all the classes in the whitelist would pass the Immutability checker (for example, an Integer's value field is not `final`), but they exhibit the necessary immutability properties. We also add `java.lang.String` to our immutable whitelist; we believe that it also satisfies the immutability criteria.

Arrays

Arrays have many uses as compile-time constants, particularly as lookup tables for decryption functions. However, supporting them in Java requires extra work since the entries of a Java array can be modified at any time. For an array variable to be a compile-time constant requires that the variable reference can't change, the constituent element references can't change, and each item should be immutable. Enforcing and checking the first and last conditions is relatively simple: the array must be declared final and its base type must implement the `Immutable` interface. However, this does not prevent the array from being modified; an element or can be updated with a different value.

To solve this, we must make sure the array's elements are not changed after initialization time. This can happen when the array or its element is used as an l-value in an assignment expression. If this occurs after initialization, this indicates an element of the array is being overwritten. The checker looks for compile-time constant arrays used inside l-values flags and them as errors.

In Java, it is possible to alias an array or a subarray to a different variable. If such aliases were made of the array, a naïve checker would miss mutations of the array by way of the alias. This risk is prevented by requiring that all occurrences of the array variable aside from its declaration occur within expressions that index the array to its full depth. We view passing partial index values explicitly as an acceptable alternative to using a partially indexed array. The other use of partially indexing arrays is when reading the length field of a subarray. This represents a legitimate case where the array is not fully indexed; given the frequency of this coding paradigm, we make a special case exception to allow partial indexing of an array only when the length field is being accessed. Thus, referring to a compile-time constant array as a whole or partially indexing a multidimensional compile-time constant array without accessing its length field is flagged as an error by our checker. This analysis requires a “closed-world” assumption, i.e. that the full source code of the program is present in order for this reasoning to be sound. If there were unchecked code present in the system, it could bypass these restrictions and modify the array.

Initializers

Not only must a compile-time constant be Immutable, but it must also be initialized to the same value every time. This means that its initializer expression should be a deterministic function, i.e. it must be environment-free. In the course of making the compile-time constant checks, the checker generates a queue of all variable initializers for compile-time constants. These will later be checked just by the environment-free checker, and which treats them as methods with no arguments. Since all compile-time constants must be final, a compile-time constant that doesn’t have a variable initializer must be initialized in a static initializer block. These too must be environment-free, and thus are added to the list of environment-free methods as they are encountered.

6.4.4 Environment-free methods

As discussed in Section 6.3.2, an environment-free method may only call a method if it is environment-free. Additionally, an environment-free method must not access global variables that are not compile-time constants.

Constructors

Constructors are treated like any other method, and any constructor that is invoked due to a new object instantiation from within an environment-free method must itself be considered environment-free. Thus, any methods that the constructor invokes must be checked for environment-freeness. This includes chained constructors or any superclass constructors that may be invoked implicitly.

Overridden methods

A class can only override an environment-free method with an environment-free method. If this were not the case, invoking the method on the base class could actually invoke the overridden method when the runtime type differs from the static type of the object. If at static analysis time, the method is deemed to be environment-free, we must ensure that the runtime method is also environment-free. Effectively, the environment-free attribute is a part of the method's signature that must be inherited with any overridden methods. The checker verifies this property. In the general case, this requires the whole program to be present. (Alternately, we could require environment-free methods to be final, but we already require a closed world for our treatment of compile-time constant arrays.)

Whitelist

Library methods called by an environment-free method require special care. In general, the checker does not have the source code to such methods so it cannot assess whether they are environment-free or not. The conservative action in this case would be to flag all calls to a library from an environment-free method as errors.

However, excluding all library functions is not practical given the size and utility of the Java library. Forbidding environment-free functions from using the large subset of the library that is environment-free unfairly constrains the programmer and represents a serious usability burden.

Whitelisted method and constructor signatures
`byte[] java.lang.String.getBytes()`
`java.lang.String(byte[])`
`java.lang.String(char[])`
`boolean java.lang.String.equals(java.lang.Object)`
`void java.lang.System.arraycopy(java.lang.Object, int, java.lang.Object, int, int)`
`boolean java.util.Arrays.equals(byte[], byte[])`
`java.util.ArrayList()`
`java.lang.Object[] java.util.ArrayList.toArray(java.lang.Object[])`
`java.lang.Object java.util.ArrayList.get(int)`
`boolean java.util.ArrayList.add(java.lang.Object)`
`int java.util.ArrayList.size()`
`java.lang.IllegalArgumentException(java.lang.String)`

Table 6.2: List of methods that are on the environment-free whitelist. We analyzed these library methods to guarantee they honor the environment-free properties.

We get around these limitations by allowing the programmer to specify a list of allowable library methods and constructors. The programmer specifies this as a list of method signatures, as in Table 6.2. Of course, it is critical that the programmer verify that the methods specified in the whitelist are in fact environment-free; doing otherwise would compromise the analysis. Table 6.2 represents the whitelist table we used for the applications we discuss in Sections 6.5.1 and 6.5.2.

For method invocations to a library function, the checker consults the whitelist to see if the function is environment-free.

Exceptions

As Table 6.5.2 shows, we have whitelisted the `IllegalArgumentException` constructor. As is a common Java coding practice, functions that receive unexpected or abnormal arguments often create and then throw this exception to the caller. The code we analyzed makes use of this pattern.

This raises the question of how to treat environment-free functions that may throw exceptions. The most natural approach is to treat such exceptions as another form of return value. Under this view, if an environment-free method is passed input that causes an exception to be thrown, the exception must be the same under all invocations. However, `java.lang.Throwable`, the root of all exceptions, contains a stack trace. This stack trace will naturally vary depending upon the caller's location in the method-call stack of the execution. Under a strict view of environment-freeness, this should not be allowed, since the return value (the exception) depends upon something other than the method's arguments. Ultimately, the source of this problem arises whenever an exception is constructed, even if it is not thrown. It is the *creation* of an exception is not environment-free, as the stack trace is filled in by `Throwable`'s constructor in a manner that depends on the stack rather than its arguments. In addition to throwing the non-deterministic exception, there is a second risk: an environment-free method could potentially examine an exception's stack trace and return a value that depends on the trace. This can be addressed simply by ensuring that the `getStackTrace()` and `printStackTrace()` methods (and any methods that call them) are not on the environment-free whitelist. The non-determinism is then hidden from the environment-free function.

We see four possible approaches for dealing with this source of non-determinism:

1. We could disallow environment-free functions from throwing exceptions. On an error, such a function could return an error code, for example. This would hinder error handling and represents a change in style. The biggest problem with this approach, however, is the ubiquity of exceptions in Java. Many library calls (which might otherwise be environment-free) can throw exceptions, particularly runtime exceptions which need not be declared. Various constructs of the language itself such as null-pointer dereferences, array index violations, and arithmetic can throw exceptions. Ensuring that Java code is exception-free is difficult and overly restrictive.

2. An alternative would be to only throw pre-existing exceptions stored in compile-time constants. For each type of exception that we wish to throw, we could initialize a compile-time constant with an exception of that type. An environment-free function that encounters an exceptional condition can throw this constant and does not need to instantiate a new exception. Since the stack trace is filled in during the exception's creation, the stack trace will not vary based on the environment-free function's call stack. Unfortunately, this approach is at least as problematic as the first one. We still need to enumerate all possible exceptions that could

be thrown by a function. Additionally, compile-time constant initializer expressions are supposed to be environment-free, but Exception creation is not, even when called from a static initializer. The stack trace depends upon class load order, which could vary depending upon the behavior of non–environment-free code.[3]

3. A third option would be to wrap calls to environment-free entry-point functions so that all `Throwable`s are caught and something else is returned. The easiest way to do this would be to return a null reference, as null is a valid value for any object type. This would keep the library's control flow and exception handling the same at the cost of losing debugging information. While this option is feasible, the loss of information and need to modify the program make this unattractive.

4. One could "define away" the problem by allowing the return value of an environment-free function to depend on its method-call stack, i.e. by treating these method calls as an implicit argument to the method. One must be careful not to relax too far, however, or environment-freeness ceases to mean much. If the function can have arbitrary dependencies on the stack, we can no longer derive the properties we want. Its dependency on the stack must be limited so that it allows for the use of exceptions but does not allow for harmful nondeterminism.

We chose a variant of the last option. We allow the return value of an environment-free function to depend on its execution stack *only* in the stack trace of any throwables it returns or throws. This is the semantics that results from allowing the construction of exceptions (and encountering exceptions resulting from method calls and language operations) but disallowing any querying of the stack traces contained within such exceptions. Adherence to this rule relies only on

[3]The stack trace includes the context of the field access or method call that referenced the class being statically initialized and thus caused it to be loaded.

ensuring that the whitelisted methods don't allow access to the stack traces of throwables; we have verified that this is the case.

6.4.5 Implementation

We implement our checker as an Eclipse 3.2.1 [1] plugin to check Java 1.4 source code. The checker is 1199 lines of code. We rely on Eclipse's visitor functionality to perform our analysis. The visitation functionality allows the checker to rely on Eclipse for parsing, name and type resolution, and walking over the typed AST. Our checks were simple enough that we did not need a data-flow engine; analysis simply consists of several visitation passes over the AST of a program.

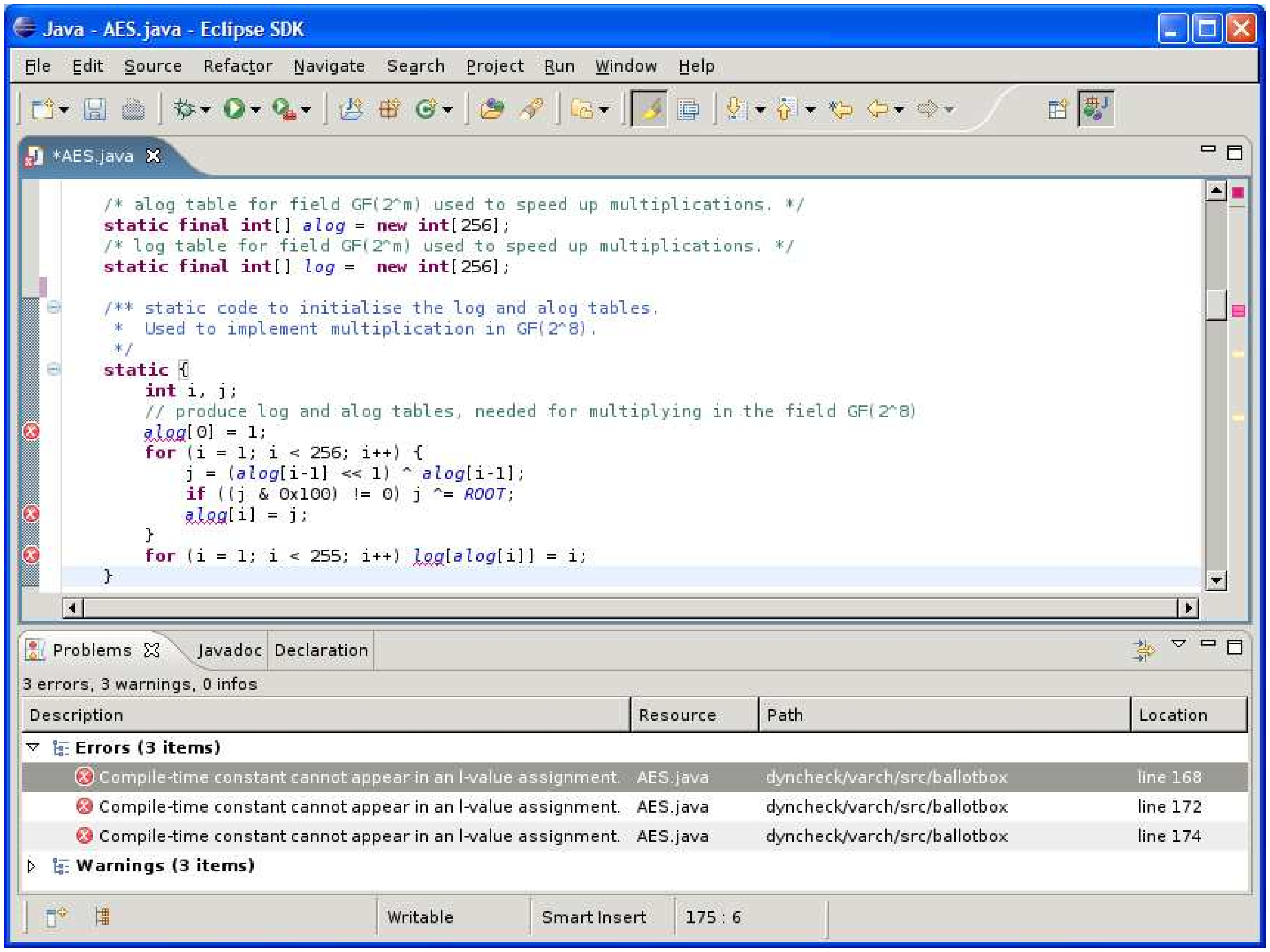
Figure 6.1 shows an image of the plugin running under Eclipse on an AES implementation. In Section 6.5.1, we discuss the results of the analysis.

6.5 Results and Discussion

We tested our checker on two applications. The tests were meant to show that the checker can find real bugs in real code as well as to verify useful properties about interesting programs. In this section, we discuss the results of running our checker as well as additional issues regarding non-determinism.

6.5.1 AES block cipher

We analyze an AES block cipher implementation to ensure that the cipher will be able to decrypt the ciphertext to the original plaintext at some later time. We analyze a third-party AES implementation [10] and check that its decryption method is environment-free. This property



Java - AES.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
*AES.java
Problems
Javadoc
Declaration
Writable
Smart Insert
175 : 6

```
    /* alog table for field GF(2^m) used to speed up multiplications. */
    static final int[] alog = new int[256];
    /* log table for field GF(2^m) used to speed up multiplications. */
    static final int[] log =  new int[256];

    /** static code to initialise the log and alog tables.
     *  Used to implement multiplication in GF(2^8).
     */
    static {
        int i, j;
        // produce log and alog tables, needed for multiplying in the field GF(2^8)
        alog[0] = 1;
        for (i = 1; i < 256; i++) {
            j = (alog[i-1] << 1) ^ alog[i-1];
            if ((j & 0x100) != 0) j ^= ROOT;
            alog[i] = j;
        }
        for (i = 1; i < 255; i++) log[alog[i]] = i;
    }
```

3 errors, 3 warnings, 0 infos

Description	Resource	Path	Location
Errors (3 items)			
Compile-time constant cannot appear in an l-value assignment.	AES.java	dyncheck/varch/src/ballotbox	line 168
Compile-time constant cannot appear in an l-value assignment.	AES.java	dyncheck/varch/src/ballotbox	line 172
Compile-time constant cannot appear in an l-value assignment.	AES.java	dyncheck/varch/src/ballotbox	line 174
Warnings (3 items)			

Figure 6.1: Screenshot of the environment-free checker detecting errors in AES code. The constants array tables `log` and `alog` are generated at class load time. This represents a modification to a compile-time constant array; we eliminate the static code block, and instead use variable initializers. After these modifications, the checker did not find any errors.

guarantees, for example, that if the cipher is used to encrypt data, it is guaranteed to be recoverable using the decrypt function and the key. We checked its 876 lines of Java source code. We added a check function, including one annotation:

```
/** @envfree */
static boolean check (byte[] plaintext,
                      byte[] encr, byte[] key) {
    AES aes = new AES();
    aes.setKey (key);
    return Arrays.equals (aes.decrypt(encr),
                          plaintext);
}
```

For the above check to guarantee decryption will be the same at some later time, the `check()` function must be environment-free, which is indicated with the annotation. The checker detected three errors, as depicted in the screenshot in Figure 6.1. The errors stemmed from the decryption function relying on two static final arrays: `int[] log` and `int[] alog`. These are logarithm and anti-logarithm tables computed at class load time in a static initializer block. The environment-free checker flagged the initialization process as erroneous. To fix the errors, we replaced the code with precomputed array initializers. After this change, the checker did not report any errors. An alternative fix would be to inspect the code and note the writes were only used for initialization and to further verify that the initializer did not make any use of the static tables before their array values were initialized.

6.5.2 Serialization of voting data structures

As detailed in Section 6.1, we began thinking about proving security properties of election systems after analyzing two commercial voting systems. Further inspecting our own prototype voting system [74], we realized that manually proving serialization is not easy. Unintended bugs (or in a larger software engineering effort, possibly malicious code) can easily interfere with deserializing a particular ballot–when using long write-ins, for example. For our voting system, we wanted to rule out problems for the critical step of recovering ballots after they've been stored to disk. The voting machine code we analyzed was over 4400 lines of Java.

We wrote five check functions very similar to the ones in Section 6.4.1 and 6.5.1, each with one annotation, and inserted calls to the check functions after serialization. The data structures being serialized include the `ballot`, as well as four message structures that are serialized and passed via serial cable to different components of the voting machine that are physically separated for isolation. It is critical that each of the serialized structures be deserialized into an equivalent copy. The inserted dynamic checks guarantee this.

The checker did find one class of errors which required us to slightly restructure the code. Two of the messages being passed between components included a monotonically increasing serial number to filter out duplicates. The last received serial number was stored as a static field inside the message class. Inside the deserialization class, the message's serial number is compared with the static value of the most recently received serial number; if the serial number was already received, the deserialization method would return null. The environment-free checker discovered that deserialization made use of a global field, and hence deserialization is not guaranteed to produce the same result in all instances. We view this as a success in our checker, as it found an error our manual inspection missed. To fix this, we separated the deserialization functionality from duplicate message suppression, and the checker no longer found any errors in the code.

6.5.3 Non-determinism

An environment-free function should not have access to “true” non-determinism, as it could cause it to produce different output at different times. Restricting code to ensure determinism is tricky, as it is easy to miss possible sources of nondeterminism. While our checker prevents access to sources of nondeterminism that are explicit in the program, Java provides implicit nondeterminism in other ways. Handling these in a general way is out of scope for our simple environment-free checker; we leave this as future work. However, we have implemented conservative checks that could be implemented in a more precise fashion since they suffice for our application.

The first source of implicit nondeterminism provided to a Java program is the layout of objects in memory. An object’s memory location is exposed via the default `hashCode()` and `toString()` functions defined in `java.lang.Object`. Two separate instances of an Object will occupy different memory addresses even if all their fields are the same, making the two objects otherwise indistinguishable. Because a method that uses `hashCode()` or `toString()` can tell the difference between invocations, it is not possible for any method that returns a newly constructed object to be truly deterministic. Since these methods are not on the whitelist, however, a `check()` method cannot call these methods explicitly to affect its return value. Our checker flags implicit calls to `toString()` from within an environment-free method as errors.

The virtual machine can throw a Java `Error` nondeterministically at runtime, such as an `OutOfMemoryError`. A devious function could intentionally exhaust memory by allocating large objects within a loop; when the JVM throws the memory error, it could use the loop index as a non-deterministic value in a `catch` or `finally` block. A general solution to this problem would involve engineering some minimal restrictions on the ability of environment-free methods to recover from errors. The applications we looked at do not have any `catch` or `finally` blocks within the environment-free code, and so do not pose a risk. Our current checker is overly conservative and bans all `catch` or `finally` blocks within environment-free code.

We view as out of scope detecting runtime resource exhaustion attacks that cause different return values; for example, a Decode function might return the correct result, or it might cause an `OutOfMemoryError`; further work will be needed to detect varying behavior in the face of resource pressure.

6.6 Conclusion

Environment-freeness is a useful property in proving properties about invertible functions. In this paper, we have defined environment-freeness for Java methods. We have described a simple algorithm for checking this property of Java programs and have discussed some of the issues that arise. We have implemented an environment-free checker and used it to permit runtime verification of the Inverse Property for an encryption implementation. A user of the encryption program can be confident that any encrypted data will later be recoverable given the associated key.

We have also applied the checker to ballot and wire serialization in our prototype voting system; we can now be confident that serialized data will be recoverable when it is later needed. This allows us to rule out a class of denial-of-service attacks on a voting machine, proving that a voter's ballot will not be lost due to buggy or malicious serialization code. We are encouraged by the added assurance provided by this check with a minimal annotation burden and believe that it bodes well for the potential for other practical program analyses to ensure useful properties of software systems.

Chapter 7

Related work

The ideas in this dissertation are built upon existing work in the security field. We apply known principles and original ideas to a new and socially important problem domain.

In contrast to prior efforts to secure general purpose operating systems, however, the voting application's limited scope helps tremendously. The experiences gained in designing secure operating systems apply to the voting case, but many of the problems can be simplified since we can deploy special purpose solutions. Finally, the progress of type-safe languages and better verification tools can only help in gaining confidence in our proposed architecture. In this chapter, we detail the relationships with existing work.

7.1 Voting

Peter Neumann was one of the earliest to write about both the security and non-security goals of electronic voting systems [62]. It is remarkably prescient in cautioning against a number of flaws in real DRE voting systems, such as vulnerabilities in using fixed passwords that later afflicted Diebold voting machines [18, 42, 72]. In a landmark study, Kohno et al. obtained Diebold source code for one of their voting machines and found many of the aforementioned basic security flaws. Their work, as well as the others detailing voting machine flaws, impelling many in the security community to seek alternatives. A review by by the California's Voting Systems Technology Assessment Advisory Board (VSTAAB) found a number of classic security bugs in Diebold election software, though they noted the impacts of the bugs could be mitigated through procedural controls [90]. This study, along with one by Feldman et al. [25], noted the importance of physical security and the breakdown of all security guarantees in its absence. Another study commissioned by the Florida State Division of Elections analyzed the source code of an ES&S voting machine in response to 18000 undervotes in a Congressional race [94]. They found numerous bugs, but could not attribute the bugs they discovered to the undervote; additionally, they commented on the difficulty in proving the absence of security problems in that particular system.

Diebold, Hart Intercivic, Sequoia Voting Systems, and Election Systems and Software each currently market DRE voting machines. Their systems feature an interface to input votes (all but the HART use a touch-screen) and store the votes internally in an electronic form. These machines have alternate interfaces to allow people with certain disabilities to vote and preserve their confidentiality. A criticism of the machines, however, is that people must trust the software running on the machines since the voter cannot be sure their vote was properly recorded. They have little reason to trust the software, in contrast to our effort where verification gives some assurance as to the software's proper design. Rebecca Mercuri has called for vendors to augment DRE machines with a Voter Verified Paper Audit Trail (VVPAT) [50, 51]. In this DRE variant, the voter must approve a paper copy of their selections that serves as the permanent record. The paper copy is typically held behind glass so the voter cannot tamper with it. The voter can verify that the paper copy accurately reflects the voter's selection, even if the software is malicious. In part because of Mercuri's efforts, jurisdictions are requiring VVPAT-enabled DRE machines. All four vendors have such machines available for sale. VVPAT helps to guarantee election integrity, but does not protect a voter's confidentiality. VVPAT may in fact negatively affect confidentiality. The techniques we look at, therefore, will be useful even in VVPAT DREs and in fact would provide a useful starting point for such an architecture. They would additionally help guarantee the correctness of the electronic copy.

Soon after the 2000 federal election, Caltech and MIT launched a project to investigate voting technology. As part of their research, they proposed an architecture for voting that they call FROGS [11]. The voter chooses their selections on an untrusted terminal, which then stores the provisional ballot onto an electronic memory device called a *frog*. The voter then takes the frog to a trusted device to verify the contents of the frog, and if they are acceptable, to cast the frog into a final ballot. The trusted device displays the frog's contents. The user can then reject the process to start over or cast the ballot, which causes the trusted device to 1) "freeze" the frog and prevent further changes; 2) send a copy of the data to a number of independent vote storage units; and 3) physically keep the frozen frog for later audits. The FROG architecture inspired some of our efforts to isolate the vote selection process, where much of the complexity lies and not require it to be part of the TCB. We also add the goal of developing an architecture designed explicitly for verification.

The SAVE voting system (Secure Architecture for Voting Electronically) is another Caltech/MIT voting proposal [79]. They propose an architecture with four components with well-defined interfaces: a user interface component, a cryptographic signature generator, a registration verifier, and a tallier. Then, they propose implementing each component except the user interface using n version programming. Each version would be implemented by a different vendor, so that corrupting the system would require corrupting each separate version. Finally, they propose using n-separate digital cameras that would take pictures of the final confirmation screen to act as a check on the user interface component. Their prototype is implemented in Java, with the components communicating via XML over sockets. n-version programming is compelling, but it increases development and testing costs; additionally, a bug in any of the versions could cause a privacy violation. The separate versions are most useful in guarding against integrity errors. Nevertheless, some of their techniques may be applicable to our setting.

The Spanish company scytl produced a whitepaper describing their proposal for securing DREs that they call Pnyx.DRE [2]. Their proposal, like the FROGS proposal, seeks to remove the vote selection module from the trusted computing base of the voting system. They propose attaching a separate device to the DRE that combines a display with a processor and two external buttons: “Confirm” and “Cancel”. The separate device is connected to the DRE using a USB or serial cable. After the voter makes their selections on the DRE, the device displays the choices to the voter, who can either confirm or cancel the selection. If the voter accepts the selection, the device encrypts the vote and adds integrity protection. The encrypted and protected ballot is then sent back to the DRE for storage. The device may optionally emit a receipt containing a random number that is part of the encrypted vote. The receipt doesn’t reveal the voter’s selection in any way. This proposal, like FROGS and our proposal, tries to engender greater trust in the election system by minimizing the TCB. However, it doesn’t help with the privacy goals of our system, since the DRE can still leak votes; it does, however, help to guarantee integrity of the votes. A study considering the machines for integration with Diebold DREs noted that the prototype, while well designed, did not completely implement the advertised specification [83].

The Dutch water board recently used a system called Rijnland Internet Election System (RIES). The system allows voters to vote over the Internet. Before the election starts, election officials generate a key k_i for each voter; for each voter i, the officials create and record a string $H_{k_i}(\text{election id})||H_{k_i}(\text{candidate 1})||\cdots||H_{k_i}(\text{candidate } n)$. The officials use an out of band paper channel, such as the postal system, to deliver the voter specific key. The officials then destroy the voter specific key. During the election period, the voter visits the election website, enters their key k_i from the mail, and then makes their selection. The voter's browser then computes and sends $H_{k_i}(\text{election id})||H_{k_i}(\text{candidate index})$. The voter can verify their proper selection was recorded by visiting the website; the election officials tally the votes by looking up the voter's selection in the list of candidates specific to the voter. The system, however, suffers from the list of flaws that Jefferson et al. noted that any Internet voting scheme suffers: a reliance on the DNS systems, lack of privacy, vulnerability to denial of service attacks, and susceptibility to worms surreptitiously changing a voter's selection and even subsequent verification [35, 34]. Hence, this approach may bring convenience but seems to sacrifices too much in the way of security for use in government elections.

In Chapter 3, we analyzed two existing cryptographic voting schemes [60, 19, 39]. Moran and Naor have produced follow on work that is based on Neff's general approach [56]. It provides integrity protection and preserves privacy even from computationally unbounded adversaries that have access to the bulletin board. They rely on a special property of Pedersen commitments, and then generalize their results to general commitment schemes. As with Neff's scheme, the use of a bulletin board invites privacy vulnerabilities.

There are other cryptographic voting protocols, but they unfortunately are not nearly as complete as Neff's or Chaum's: they remain protocols and are not yet systems. For example, Josh Benaloh presents an outline of two cryptographic approaches, one similar to the FROGS system [8]. However, as we showed in Chapter 3, there is a large gap between protocol and a system, and that gap can often impact security. A second lesson is that the cryptographic voting protocols cannot treat humans as perfect actors, as is typical in traditional security protocols: a person will make mistakes and may not follow their end of the protocol. Attackers can take advantage of this fallibility to erode a voter's privacy or steal their vote.

Ka-Ping Yee et al. designed a voting system using pre-rendered user interfaces to also minimize the amount of trust in a voting system [95]. He uses a data structure similar to a deterministic finite state machine with the user's input controlling the transitions between states of pre-rendered ballot images. The pre-rendered ballot images eliminate UI toolkits and a large part of the application and OS complexity from the voting machine. Yee's prototype is written in fewer than 300 lines of Python, making manual verification of the software a possibility.

Work in conjunction with Molnar et al. described algorithmic and hardware techniques to store votes on a programmable read-only memory device [55]. Their storage mechanism was meant to preserve anonymity through a history independence property and by eliminating subliminal channels in the storage format, while retaining the ability to detect tampering with the storage media after polls have closed. Follow on work has eliminated the need for special hardware by using cryptographic techniques [9].

7.2 Information Flow

One of the techniques we leverage is managing the flow of confidential information within the application: if a component cannot see confidential information it cannot leak it. This principle of guarding information flow based on principals has been more generally studied in the context of multilevel security (MLS) [77]. Multilevel security systems manage data sources with different secrecy labels (e.g. unclassified, secret, top secret) and ensure that the programs that interact with these data sources also honor the secrecy labels.

The LOCK program from SRI tried for 17 years to build a MLS system. They originally intended to use a separate processor called the SIDEARM as a reference monitor [76]. The LOCK program had its roots in the PSOS (Provably Secure Operating System) project [24, 63]. They faced problems with their hardware based reference monitor since it added cost and time to completion. Additionally, the LOCK designers intended to write formal specifications and ensure their correctness with the GYPSY proof checker. An important realization of their effort was that GYPSY was not sophisticated enough and ultimately did not help in detecting bugs. This cautionary tale about the difficulty in formal verification steered our efforts towards architectures to simplify verification instead of work on formal tools. The exercise was not a waste, however, since they found that the time spent to consider the formalisms and prepare the specifications led the designers themselves to catch bugs they believe they would have otherwise missed. There are important differences, however; they were trying to build a general purpose system, while we are designing a specific one. Additionally, formal methods have advanced greatly in the intervening years, and as we show, can be used to achieve successes.

The Starlight Interactive Link is a hardware device that allows a workstation trusted with secret data to safely interact with an unclassified network [4]. As in the MLS scenario, a chief concern is secret data leaking onto the untrusted network. The Starlight Interactive Link acts as a data diode. Their data diode serve a similar purpose to the buses we introduce in Chapter 5. The enable a system to be partitioned but allow limited communication topologies enforced by hardware. Users typically install the Interactive Link between the trusted workstation and the keyboard and mouse. The Interactive Link routes the I/O either to the trusted workstation or to a remote untrusted application server connected to the untrusted network. Additionally, the Interactive Link can display the untrusted application on the secure workstation by simulating parts of the X protocol to enforce the one way flow of information from low to high. Subsequent work builds on the Interactive Link and extends the technique of data diodes in software to develop MLS CORBA for distributed applications [38].

Jones and Bowersox develop a one-way serial bus to prevent backwards data-flow and mitigate the presence of covert channels [37]. It is designed for easy verification: the entire data diode fits on a small PCB and is comprised of fewer than 20 discrete components. We could imagine using their verifiable one-way serial bus in our architecture and provides a real world example that it is possible to build the communication primitives necessary to enable communication in a verifiable topology. The intent is that it should be simple to verify the correctness of the specification and further that the physical data diode matches the specification. The idea of simple verification parallels our effort. Jones and Bowersox postulate the data diode is most useful to publish incremental results from the tallying computer to public reporting software that may have an Internet connection.

7.3 Isolation

Another key technique we leverage is the principle of isolation; it is well understood that restricting different parts of a system to communicate through well-defined standard interfaces may increase security. A vulnerability in one component can only affect another component if an adversary can subvert the first and then exploit the interface between components. Recognizing this fact, the systems community has studied and implemented many different isolation mechanisms. We make use of this idea in this work.

High assurance applications take a similar view to the importance of verifying that systems achieve specific safety, reliability, and security properties. High assurance applications are frequently found in transportation, medical, or communications contexts. They use a wide variety of techniques. We we adopt on some of these techniques in our work, such as minimizing TCB sizes and decomposing the application into cleanly specified components. The challenge, in our case, is to elucidate just how to take these high level concepts and apply them to the voting specific context, since secure components may not be composeable into a secure system [49]. Additionally, there are important differences in the use of components and isolation in traditional high-assurance applications and voting. Deep space applications use multiple components for reliability and fault tolerance [96], while telephony communications use redundant components for software upgrades [96]. In their MLS-PCA avionics architecture, Northrop Grumman is proposing an architecture suitable to the Department of Defense's Joint Vision 2020 [91]. MLS-PCA is an architecture for future avionics systems supporting tens to hundreds of thousands of separate processors. Their use of isolation is manifold: mission flexibility, requirement of supporting multi-level security for interoperating with non-profit organizations, and a reduction in the amount of trusted software over traditional federated architectures. Of these reasons, the last, is most related to our setting in that isolation helps in minimizing the size and number of trusted components.

Operating systems originally were designed to provide safe resource sharing, and in some cases, isolation among different processes. Unfortunately, many people have noted they do not provide sufficient isolation: the TCB is large and complex, and bugs allow one process to subvert another. Microkernels such as Mach or L4 try to place each module in separate protection domains, but have not ultimately caught on, in part due to the performance overhead [47, 69]. A more recent lightweight approach is to run device drivers in their own protection domain [86, 87]. Swift et al. note that drivers are a significant source of bugs, and they believe that driver isolation can help improve reliability and reduce the time between rebooting the operating system.

There is also a trend to using virtual machine monitors or similar techniques to provide isolation since bugs in operating systems and side channels do not provide enough separation. Disco, Denali, and Xen are virtual machine monitors[1] that can be used to provide isolation [6, 14, 92, 93]. This approach was first suggested by Rushby where he suggested that a *security kernel* should provide the same isolation that an application sees if it was hosted on separate systems [71]. Rushby points to the VM/370 as a virtual machine monitor that begins to provide the isolation level demanded by a security kernel. With the drop in hardware prices, it becomes feasible in this dissertation to allocate a separate microprocessor to each component. Rushby used the security kernel as a mechanism to help securely multiplex hardware, while I use separate hardware.

Provos et al. noticed that in many privileged applications, two separate protection domains, representing high and low, can be used to help isolate vulnerabilities in the low domain from

[1] Xen is actually a paravirtualizing virtual machine monitor since it exports an instruction set that is slightly different than found in the underlying hardware implementation.

letting the attacker achieve the high component's privileges [67]. Their response was to privilege separate OpenSSH, so as to minimize the trusted code; they showed it was feasible to do so with a minimal performance penalty. This approach required hand-modified code. Brumley et al. automate the separation task using static analysis and source to source translation tools [12].

Many of these systems differ from our intended isolation uses in that they are meant for the general purpose systems and hence general purpose applications. Our use is a specialized application: we can therefore create customized solutions that may not be feasible in the general case. We can design and use hardware tailored to the voting application that are not useful for general purpose applications.

7.4 Verification

Hoare first broached the idea that analysis tools may be able to prove properties about programs given a few starting axioms [30]. His work posited that it may be possible to write proofs for the correct functioning of programs. With many advances in the field, it is now possible to prove specialized properties of an application given the source code.

Verification tools were able to generate proofs for the MLS properties in the KSOS-6 operating system [84]. KSOS-6 has 10,000 lines of source and 3300 lines of a formal specification language called SPECIAL; their MLS verification tool detected 33 security flaws. However, 29 of the flaws were detected manually. The considerable effort spent in generating the specification and working with the verification tools points to the immaturity of the tools. Another report candidly remarks that "formal methods are useful only when the developer can pose the right questions" and it may not be possible to generate complete proofs of correctness in large, general purpose operating systems [78].

A more recent success story verifies the containment mechanism in the EROS operating system [82]. EROS is a capability based operating system, and they were able to verify the OS's *containment* mechanism, whereby the operating system creates a restricted environment with a limited set of capabilities. They demonstrate that the restricted environment can only access the resources granted by its capability set and no others.

Joe-E is a subset of Java that enforces the capability discipline [53]. We drew inspiration for the environment-free checker from their work; they provide a useful framework for immutability that we use as the basis for the environment-free checker's compile time constants.

It is now possible to soundly detect all format string vulnerabilities in C code [81] and find all user-kernel bugs in the Linux kernel [36]. Both techniques rely on type inference, a technique for developers to add a few annotations to the type system and then perform analyses to detect inconsistencies in the enriched type system, which are possible bugs in the application software. These techniques show the promise of being able to prove real security properties about real code.

Spec# [7] and JML [15, 44] are language extensions that allow the programmer to specify pre-conditions and post-conditions on methods as well as invariants for classes for the C# and Java language respectively. They followed Bertrand Meyer's work where he suggested that classes and methods should have a contract specified through annotations [54]. Using these extra annotations, program verifiers check that the code is consistent with the specification. These tools provide a first step in proving systems correct.

Additionally, it should be mentioned that safe languages, such as Java or C#, eliminate a large class of vulnerabilities since the virtual machine in which they run enforces the type-safety of the code it executes. We take advantage of these features to ease the verification task since the language itself does not allow for programs with certain vulnerabilities to be considered valid.

7.5 State management

The Recovery Oriented Computing (ROC) project advocates a unique view to state management [65]. The project seeks to increase reliability and availability of software services; as a part of this, they suggest that components in a software application should be designed for reboot [16, 17]. Each component should be able to be restarted at any time, and in fact they call for prophylactic reboots to reset state in volatile member variables, based in part by work by Huang et al. [31]. In order for a component to be rebootable, it needs to store all persistent state in a separate module and not hold any pointers across component boundaries. Our work also uses rebootable components, but for a different purpose: security. A voter who knows that a component reboots after leaving the voting booth can be better assured that their sensitive information cannot leak to the next voter if there is no way for sensitive information to leave the ballot box; secondly, a voter who knows that the voting machine reboots before they arrive to use it can be better assured that the previous voter's actions will not affect their voting session.

Chapter 8

Conclusion

In this dissertation, we have explored a property based approach to improving voting security. Under this view, one must be cognizant of how endowing a voting system with one property impacts the system's goals. It is important, also, to consider the voting system as a whole, including the technology as well as the humans that interact with the technology: the technology does not exist in a vacuum.

Our solutions apply to a range of voting platforms and address different properties. Rebooting can be used as an effective approach to stem privacy violations across voter sessions for a variety of different voting technologies. Likewise, our componentised voting architecture applies to DRE based systems to more easily prove a few voting properties. Our software analysis techniques can prove deserialization and decryption are correct in a fail-stop model. These analyses are useful for all voting platforms, and can even apply in non-voting contexts.

People should be able to trust their voting technology has sufficient security guarantees. The fully verified voting machine is not yet in our grasp. But this should not stop us from attempting to design and build voting systems that meet increasingly more security properties. This dissertation begins that path towards the verified voting machine.

Bibliography

[1] The Eclipse Platform. `http://www.eclipse.org`.

[2] Auditability and voter-verifiability for electronic voting terminals. `http://www.scytl.com/docs/pub/a/PNYX.DRE-WP.pdf`, December 2004. White paper.

[3] Atul Adya, William Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John Douceur, Jon Howell, Jacob Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symposium on Operating System Design and Implementation (OSDI)*, pages 1–14, December 2002.

[4] M. Anderson, C. North, J. Griffin, R. Milner, J. Yesberg, and K. Yiu. Starlight: Interactive Link. In *Proceedings of the 12th Annual Computer Security Applications Conference (ACSAC)*, 1996.

[5] Jonathan Bannet, David W. Price, Algis Rudys, Justin Singer, and Dan S. Wallach. Hack-a-vote: Demonstrating security issues with electronic voting systems. *IEEE Security and Privacy Magazine*, 2(1):32–37, Jan./Feb. 2004.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Time Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Sstems Principles (SOSP 2003)*, October 2003.

[7] Mike Barnett, K. Rustan Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, 2004.

[8] Josh Benaloh. Simple verifiable elections. In *USENIX/ACCURATE Electronic Voting Technology Workshop*, October 2006.

[9] John Bethencourt, Dan Boneh, and Brent Waters. Cryptographic methods for storing ballots on a voting machine. In *14th Annual Network & Distributed System Security Conference (NDSS 2007)*, February 2007.

[10] Lawrie Brown. AEScalc. `http://www.unsw.adfa.edu.au/~lpb/src/AEScalc/AEScalc.jar`.

[11] Shuki Bruck, David Jefferson, and Ronald Rivest. A modular voting architecture (“Frogs”). `http://www.vote.caltech.edu/media/documents/wps/vtp_wp3.pdf`, August 2001. Voting Technology Project Working Paper.

[12] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[13] Jeremy Bryans and Peter Ryan. A dependability analysis of the Chaum digital voting scheme. Technical Report CS-TR-809, University of Newcastle upon Tyne, July 2003.

[14] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[15] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joseph Kiniry, Gary Leavens, K. Rustan Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[16] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workship on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[17] George Candea, Shinishi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – a technique for cheap recovery. In *6th Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[18] RABA Innovative Solution Cell. Trusted agent report Diebold AccuVote-TS voting system, January 2004.

[19] David Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy Magazine*, 2(1):38–47, Jan.–Feb. 2004.

[20] David Chaum, February 2005. Personal Communication.

[21] CIBER. Diebold Election Systems, Inc. Source code review and functional testing. California Secretary of State's Voting Systems Technology Assessment Advisory Board (VSTAAB), February 2006.

[22] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, October 2001.

[23] Dawson Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, October 1995.

[24] Richard Feiertag and Peter Neumann. The foundations of a Provably Secure Operating System (PSOS). In *Proceedings of the National Computer Conference*, pages 329–334, 1979.

[25] Ariel Feldman, J. Alex Halderman, and Edward W. Felten. Security analysis of the Diebold AccuVote-TS voting machine. In submission.

[26] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984.

[27] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(2):270–299, 1984.

[28] Nevin Heintze and J. D. Tygar. A model for secure protocols and their compositions. *IEEE Transactions on Software Engineering*, 22(1):16–30, January 1996.

[29] Gernot Heiser. Secure embedded systems need microkernels. *USENIX ;login*, 30(6):9–13, December 2005.

[30] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[31] Yennun Huang, CHandra Kintala, Nick Kolettis, and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 1995.

[32] Markus Jakobsson. A practical mix. In *Advances in Cryptology – EUROCRYPT 1998*, volume 1403 of *Lecture Notes in Computer Science*, pages 448–461. Springer-Verlag, May/June 1998.

[33] Markus Jakobsson, Ari Juels, and Ronald Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *11th USENIX Security Symposium*, pages 339–353, August 2002.

[34] David Jefferson, Aviel Rubin, Barbara Simons, and David Wagner. Analyzing Internet voting security. *Communications of the ACM*, 47(10):59–64, October 2004.

[35] David Jefferson, Aviel Rubin, Barbara Simons, and David Wagner. A security analysis of the secure electronic registration and voting experiment (SERVE). `http://www.cs.berkeley.edu/~daw/papers/servereport.pdf`, January 2004. Report to the Department of Defense (DoD).

[36] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[37] Douglas Jones and Tom Bowersox. Secure data export and auditing using data diodes. In *USENIX/ACCURATE Electronic Voting Technology Workshop*, October 2006.

[38] Myong Kang, Judith Froscher, and Ira Moskowitz. An architecture for multilevel secure interoperability. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC 97)*, 1997.

[39] Chris Karlof, Naveen Sastry, and David Wagner. Cryptographic voting protocols: A systems perspective. In *Fourteenth USENIX Security Symposium (USENIX Security 2005)*, August 2005.

[40] Arthur Keller, David Mertz, Joseph Hall, and Arnold Urkin. Privacy issues in an electronic voting machine. In *ACM Workshop on Privacy in the Electronic Society*, pages 33–34, October 2004. Full paper available at `http://www.sims.berkeley.edu/~jhall/papers/`.

[41] Paul Kocher and Bruce Schneier. Insider risks in elections. *Communications of the ACM*, 47(7):104, July 2004.

[42] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, pages 27–40, May 2004.

[43] Markus Kuhn. Optical time-domain eavesdropping risks of CRT displays. In *IEEE Symposium on Security and Privacy*, May 2002.

[44] Gary Leavens and Yoonsik Cheon. Design by contract with JML. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`.

[45] Matt Lepinski, Silvio Micali, and abhi shelat. Collusion-free protocols. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, May 2005.

[46] Matt Lepinski, Silvio Micali, and abhi shelat. Fair zero knowledge. In *Proceedings of the 2nd Theory of Cryptography Conference*, February 2005.

[47] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70, September 1996.

[48] Heiko Mantel. On the composition of secure systems. In *IEEE Symposium on Security and Privacy*, pages 88–101, May 2002.

[49] Daryl McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, May 1988.

[50] Rebecca Mercuri. *Electronic Vote Tabulation Checks & Balances*. PhD thesis, School of Engineering and Applied Science of the University of Pennsylvania, 2000.

[51] Rebecca Mercuri. A better ballot box? *IEEE Spectrum*, 39(10):46–50, October 2002.

[52] David Mertz. XML Matters: Practical XML data design and manipulation for voting systems. `http://www-128.ibm.com/developerworks/xml/library/x-matters36.html`, June 2004.

[53] Adrian Mettler and David Wagner. The Joe-E language specification (draft). Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley, March 17 2006.

[54] Bertrand Meyer. Applying “Design by contract”. *IEEE Computer*, 25(10):40–51, 1992.

[55] David Molnar, Tadayoshi Kohno, Naveen Sastry, and David Wagner. Tamper-evident, history-independent, subliminal-free data structures on PROM storage -or- How to store ballots on a voting machine (extended abstract). In *IEEE Symposium on Security and Privacy*, May 2006.

[56] Tal Moran and Moni Naor. Receipt-free universally-verifiable voting with everlasting privacy. In *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 373–392, August 2006.

[57] Deirdre Mulligan and Joseph Hall. Preliminary analysis of e-voting problems highlights need for heightened standards and testing. A whitepaper submission to the NRC's Committee on Electronic Voting, `http://www7.nationalacademies.org/cstb/project_evoting_mulligan.pdf`, December 2004.

[58] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *8th ACM Conference on Computer and Communications Security (CCS 2001)*, pages 116–125, November 2001.

[59] C. Andrew Neff, October 2004. Personal Communication.

[60] C. Andrew Neff. Practical high certainty intent verification for encrypted votes. `http://www.votehere.net/vhti/documentation`, October 2004.

[61] C. Andrew Neff. Verifiable mixing (shuffling) of El Gamal pairs. `http://www.votehere.net/vhti/documentation`, April 2004.

[62] Peter Neumann. Security criteria for electronic voting. In *Proceedings of the 16th National Computer Security Conference*, September 1993.

[63] Peter Neumann and Richard Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, 1997.

[64] Peter G. Neumann. Principled assuredly trustworthy composable architectures. Final report for Task 1 of SRI Project 11459, as part of DARPA's Composable High-Assurance Trustworthy Systems (CHATS) program, 2004.

[65] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical report, University of California, Berkeley, March 2002.

[66] Birgit Pfitzmann and Andreas Pfitzmann. How to break the direct RSA-implementation of MIXes. In *Advances in Cryptology – EUROCRYPT 1989*, volume 434 of *Lecture Notes in Computer Science*, pages 373–381. Springer-Verlag, April 1989.

[67] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[68] Mohan Rajagopalan, Saumya Debray, Matti Hiltunen, and Richard Schlichting. Automated operating system specialization via binary rewriting. Technical Report TR05-03, University of Arizona, February 2005.

[69] Richard Rashid Jr., Avadis Tevanian, Michael Young, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[70] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 1–14, March 2003.

[71] John Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, December 1981.

[72] Science Applications International Corporation (SAIC). Risk assessment report Diebold AccuVote-TS voting system and processes, September 2003.

[73] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[74] Naveen Sastry, Tadayoshi Kohno, and David Wagner. Designing voting machines for verification. In *Fifteenth USENIX Security Symposium (USENIX Security 2006)*, August 2006.

[75] Naveen Sastry, Adrian Mettler, and David Wagner. Verifying serialization through environment-freeness, 2007. In submission to PLAS 2007.

[76] O. Sami Saydjari. LOCK: An historical perspective. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*, 2002.

[77] O. Sami Saydjari. Multilevel security: Reprise. *IEEE Security and Privacy*, 2(5):64–67, 2004.

[78] Fred Schneider, editor. *Trust in Cyberspace*. National Research Council, 1999.

[79] Ted Selker and Jonathan Goler. The SAVE system – secure architecture for voting electronically. *BT Technology Journal*, 22(4), October 2004.

[80] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWAtt: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[81] Umesh Shankar, Kunal Talwar, Jeffrey Foster, and David Wagner. Detecting format-string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[82] Jonathan Shapiro and Samuel Weber. Verifying the EROS confinement mechansim. In *IEEE Symposium on Security and Privacy*, May 2000.

[83] Alan T. Sherman, Aryya Gangopadhyay, Stephen H. Holden, George Karabatis, A. Gunes Koru, Chris M. Law, Donald F. Norris, John Pinkston, Andrew Sears, , and Dongsong Zhang. An examination of vote verification technologies: Findings and experiences from the maryland study. In *USENIX/ACCURATE Electronic Voting Technology Workshop*, October 2006.

[84] Jonathan Silverman. Reflections on the verification of the security of an operating system kernel. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, December 1983.

[85] Pete Slover. Some Texas counties are clinging to the chad. Dallas Morning News, March 8 2004.

[86] Michael Swift, Muthukaruppan Annamalai, Brian Bershad, and Henry Levy. Recovering device drivers. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation*, December 2004.

[87] Michael Swift, Brian Bershad, and Henry Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Sstems Principles (SOSP 2003)*, October 2003.

[88] Wim van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4, 1985.

[89] Poorvi Vora. David Chaum's voter verification using encrypted paper receipts. Cryptology ePrint Archive, Report 2005/050, February 2005. `http://eprint.iacr.org/`.

[90] David Wagner, David Jefferson, Matt Bishop, Chris Karlof, and Naveen Sastry. Security analysis of the Diebold AccuBasic interpreter. California Secretary of State's Voting Systems Technology Assessment Advisory Board (VSTAAB), February 2006.

[91] Clark Weissman. MLS-PCA: A high assurance security architecture for future avionics. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, 2003.

[92] Andrew Whitaker, Marianne Shaw, and Steven Gribble. Denali: A scalable isolation kernel. In *10th ACM SIGOPS European Workship*, September 2002.

[93] Andrew Whitaker, Marianne Shaw, and Steven Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th ACM/USENIX Symposium on Operating System Design and Implementation*, December 2002.

[94] Alec Yasinsac, David Wagner, Matt Bishop, Ted Baker, Breno de Madeiros, Gary Tyson, Michael Shamos, and Mike Burmester. Software review and security analysis of the ES&S iVoteronic 8.0.1.2 voting machine firmware. Report commissioned by the Florida State Division of Elections,, February 23 2007.

[95] Ka-Ping Yee, David Wagner, Marti Hearst, and Steven Bellovin. Prerendered user interfaces for high-assurance electronic voting. In *USENIX/ACCURATE Electronic Voting Technology Workshop*, October 2006.

[96] I-Ling Yen and Ray Paul. Key applications for high-assurance systems. *IEEE Computer*, 31(4):35–45, April 1998.